Nim Manual 1.3.5

Andreas Rumpf, Zahary Karadjov

August 21, 2020

Contents

"Complexity" seems to be a lot like "energy": you can transfer it from the end user to one/some of the other players, but the total amount seems to remain pretty much constant for a given task. – Ran

1 About this document

Note: This document is a draft! Several of Nim's features may need more precise wording. This manual is constantly evolving into a proper specification.

Note: The experimental features of Nim are covered here.

Note: Assignments, moves and destruction are specified in the destructors document.

This document describes the lexis, the syntax, and the semantics of the Nim language.

To learn how to compile Nim programs and generate documentation see Compiler User Guide and DocGen Tools Guide.

The language constructs are explained using an extended BNF, in which `(a)*` means 0 or more `a`'s, `a+` means 1 or more `a`'s, and `(a)?` means an optional *a*. Parentheses may be used to group elements.

`&` is the lookahead operator; `&a` means that an `a` is expected but not consumed. It will be consumed in the following rule.

The `|`, `/` symbols are used to mark alternatives and have the lowest precedence. `/` is the ordered choice that requires the parser to try the alternatives in the given order. `/` is often used to ensure the grammar is not ambiguous.

Non-terminals start with a lowercase letter, abstract terminal symbols are in UPPERCASE. Verbatim terminal symbols (including keywords) are quoted with `'`. An example:

```
ifStmt = 'if' expr ':' stmts ('elif' expr ':' stmts)* ('else' stmts)?
```

The binary `^*` operator is used as a shorthand for 0 or more occurrences separated by its second argument; likewise `^+` means 1 or more occurrences: `a ^+ b` is short for `a (b a)*` and `a ^* b` is short for `(a (b a)*)?`. Example:

```
arrayConstructor = '[' expr ^* ',' ']'
```

Other parts of Nim, like scoping rules or runtime semantics, are described informally.

2 Definitions

Nim code specifies a computation that acts on a memory consisting of components called locations. A variable is basically a name for a location. Each variable and location is of a certain type. The variable's type is called static type, the location's type is called dynamic type. If the static type is not the same as the dynamic type, it is a super-type or subtype of the dynamic type.

An identifier is a symbol declared as a name for a variable, type, procedure, etc. The region of the program over which a declaration applies is called the scope of the declaration. Scopes can be nested. The meaning of an identifier is determined by the smallest enclosing scope in which the identifier is declared unless overloading resolution rules suggest otherwise.

An expression specifies a computation that produces a value or location. Expressions that produce locations are called l-values. An l-value can denote either a location or the value the location contains, depending on the context.

A Nim program consists of one or more text source files containing Nim code. It is processed by a Nim compiler into an executable. The nature of this executable depends on the compiler implementation; it may, for example, be a native binary or JavaScript source code.

In a typical Nim program, most of the code is compiled into the executable. However, some of the code may be executed at compile time. This can include constant expressions, macro definitions, and Nim procedures used by macro definitions. Most of the Nim language is supported at compile time, but there are some restrictions – see Restrictions on Compile-Time Execution for details. We use the term runtime to cover both compile-time execution and code execution in the executable.

The compiler parses Nim source code into an internal data structure called the abstract syntax tree (AST). Then, before executing the code or compiling it into the executable, it transforms the AST through semantic analysis. This adds semantic information such as expression types, identifier meanings, and in some cases expression values. An error detected during semantic analysis is called a static error. Errors described in this manual are static errors when not otherwise specified.

A panic is an error that the implementation detects and reports at runtime. The method for reporting such errors is via *raising exceptions* or *dying with a fatal error*. However, the implementation provides a means to disable these runtime checks. See the section pragmas25 for details.

Whether a panic results in an exception or in a fatal error is implementation specific. Thus the following program is invalid; even though the code purports to catch the `IndexDefect` from an out-of-bounds array access, the compiler may instead choose to allow the program to die with a fatal error.

```
var a: array[0..1, char]
let i = 5
try:
  a[i] = 'N'
except IndexDefect:
  echo "invalid index"
```

The current implementation allows to switch between these different behaviors via `-panics:on|off`. When panics are turned on, the program dies on a panic, if they are turned off the runtime errors are turned into exceptions. The benefit of `-panics:on` is that it produces smaller binary code and the compiler has more freedom to optimize the code.

An unchecked runtime error is an error that is not guaranteed to be detected, and can cause the subsequent behavior of the computation to be arbitrary. Unchecked runtime errors cannot occur if only safe language features are used and if no runtime checks are disabled.

A constant expression is an expression whose value can be computed during semantic analysis of the code in which it appears. It is never an l-value and never has side effects. Constant expressions are not limited to the capabilities of semantic analysis, such as constant folding; they can use all Nim language features that are supported for compile-time execution. Since constant expressions can be used as an input to semantic analysis (such as for defining array bounds), this flexibility requires the compiler to interleave semantic analysis and compile-time code execution.

It is mostly accurate to picture semantic analysis proceeding top to bottom and left to right in the source code, with compile-time code execution interleaved when necessary to compute values that are required for subsequent semantic analysis. We will see much later in this document that macro invocation not only requires this interleaving, but also creates a situation where semantic analysis does not entirely proceed top to bottom and left to right.

3 Lexical Analysis

3.1 Encoding

All Nim source files are in the UTF-8 encoding (or its ASCII subset). Other encodings are not supported. Any of the standard platform line termination sequences can be used - the Unix form using ASCII LF (linefeed), the Windows form using the ASCII sequence CR LF (return followed by linefeed), or the old Macintosh form using the ASCII CR (return) character. All of these forms can be used equally, regardless of platform.

3.2 Indentation

Nim's standard grammar describes an indentation sensitive language. This means that all the control structures are recognized by indentation. Indentation consists only of spaces; tabulators are not allowed.

The indentation handling is implemented as follows: The lexer annotates the following token with the preceding number of spaces; indentation is not a separate token. This trick allows parsing of Nim with only 1 token of lookahead.

The parser uses a stack of indentation levels: the stack consists of integers counting the spaces. The indentation information is queried at strategic places in the parser but ignored otherwise: The pseudo terminal `IND{>}` denotes an indentation that consists of more spaces than the entry at the top of the stack; `IND{=}` an indentation that has the same number of spaces. `DED` is another pseudo terminal that describes the *action* of popping a value from the stack, `IND{>}` then implies to push onto the stack.

With this notation we can now easily define the core of the grammar: A block of statements (simplified example):

```
ifStmt = 'if' expr ':' stmt
         (IND{=} 'elif' expr ':' stmt)*
         (IND{=} 'else' ':' stmt)?

simpleStmt = ifStmt / ...

stmt = IND{>} stmt ^+ IND{=} DED  # list of statements
     / simpleStmt                 # or a simple statement
```

3.3 Comments

Comments start anywhere outside a string or character literal with the hash character #. Comments consist of a concatenation of comment pieces. A comment piece starts with # and runs until the end of the line. The end of line characters belong to the piece. If the next line only consists of a comment piece with no other tokens between it and the preceding one, it does not start a new comment:

```
i = 0     # This is a single comment over multiple lines.
  # The scanner merges these two pieces.
  # The comment continues here.
```

Documentation comments are comments that start with two ##. Documentation comments are tokens; they are only allowed at certain places in the input file as they belong to the syntax tree!

3.4 Multiline comments

Starting with version 0.13.0 of the language Nim supports multiline comments. They look like:

```
#[Comment here.Multiple linesare not a problem.]#
```

Multiline comments support nesting:

```
#[  #[ Multiline comment in already   commented out code. ]#proc p[T](x: T) = discard]#
```

Multiline documentation comments also exist and support nesting too:

```
proc foo =
  ##[Long documentation comment  here.  ]##
```

3.5 Identifiers & Keywords

Identifiers in Nim can be any string of letters, digits and underscores, with the following restrictions:

- begins with a letter
- does not end with an underscore _
- two immediate following underscores __ are not allowed::

  letter ::= 'A'..'Z' | 'a'..'z' | 'x80'..'xff' digit ::= '0'..'9' IDENTIFIER ::= letter (['_'] (letter | digit))*

Currently any Unicode character with an ordinal value > 127 (non ASCII) is classified as a `letter` and may thus be part of an identifier but later versions of the language may assign some Unicode characters to belong to the operator characters instead.

The following keywords are reserved and cannot be used as identifiers:

```
addr and as asm
bind block break
case cast concept const continue converter
defer discard distinct div do
elif else end enum except export
finally for from func
```

```
if import in include interface is isnot iterator
let
macro method mixin mod
nil not notin
object of or out
proc ptr
raise ref return
shl shr static
template try tuple type
using
var
when while
xor
yield
```

Some keywords are unused; they are reserved for future developments of the language.

3.6 Identifier equality

Two identifiers are considered equal if the following algorithm returns true:

```
proc sameIdentifier(a, b: string): bool =
  a[0] == b[0] and
    a.replace("_", "").toLowerAscii == b.replace("_", "").toLowerAscii
```

That means only the first letters are compared in a case sensitive manner. Other letters are compared case insensitively within the ASCII range and underscores are ignored.

This rather unorthodox way to do identifier comparisons is called partial case insensitivity and has some advantages over the conventional case sensitivity:

It allows programmers to mostly use their own preferred spelling style, be it humpStyle or snake_style, and libraries written by different programmers cannot use incompatible conventions. A Nim-aware editor or IDE can show the identifiers as preferred. Another advantage is that it frees the programmer from remembering the exact spelling of an identifier. The exception with respect to the first letter allows common code like `var foo: Foo` to be parsed unambiguously.

Note that this rule also applies to keywords, meaning that `notin` is the same as `notIn` and `not_in` (all-lowercase version (`notin`, `isnot`) is the preferred way of writing keywords).

Historically, Nim was a fully style-insensitive language. This meant that it was not case-sensitive and underscores were ignored and there was not even a distinction between `foo` and `Foo`.

3.7 String literals

Terminal symbol in the grammar: `STR_LIT`.

String literals can be delimited by matching double quotes, and can contain the following escape sequences:

Strings in Nim may contain any 8-bit value, even embedded zeros. However some operations may interpret the first binary zero as a terminator.

3.8 Triple quoted string literals

Terminal symbol in the grammar: `TRIPLESTR_LIT`.

String literals can also be delimited by three double quotes `"""` ... `"""`. Literals in this form may run for several lines, may contain `"` and do not interpret any escape sequences. For convenience, when the opening `"""` is followed by a newline (there may be whitespace between the opening `"""` and the newline), the newline (and the preceding whitespace) is not included in the string. The ending of the string literal is defined by the pattern `"""[^"]`, so this:

```
""""long string within quotes""""
```

Produces:

```
"long string within quotes"
```

Escape sequence	Meaning
`\p`	platform specific newline: CRLF on Windows, LF on Unix
`\r`, `\c`	carriage return
`\n`, `\l`	line feed (often called newline)
`\f`	form feed
`\t`	tabulator
`\v`	vertical tabulator
`\\`	backslash
`\"`	quotation mark
`\'`	apostrophe
\ '0'..'9'+	character with decimal value d; all decimal digits directly following are used for the character
`\a`	alert
`\b`	backspace
`\e`	escape [ESC]
`\x` HH	character with hex value HH; exactly two hex digits are allowed
`\u` HHHH	unicode codepoint with hex value HHHH; exactly four hex digits are allowed
`\u` {H+}	unicode codepoint; all hex digits enclosed in `{}` are used for the codepoint

3.9 Raw string literals

Terminal symbol in the grammar: `RSTR_LIT`.

There are also raw string literals that are preceded with the letter `r` (or `R`) and are delimited by matching double quotes (just like ordinary string literals) and do not interpret the escape sequences. This is especially convenient for regular expressions or Windows paths:

```
var f = openFile(r"C:\texts\text.txt") # a raw string, so ''\t'' is no tab
```

To produce a single `"` within a raw string literal, it has to be doubled:

```
r"a""b"
```

Produces:

```
a"b
```

`r""""` is not possible with this notation, because the three leading quotes introduce a triple quoted string literal. `r"""` is the same as `"""` since triple quoted string literals do not interpret escape sequences either.

3.10 Generalized raw string literals

Terminal symbols in the grammar: `GENERALIZED_STR_LIT`, `GENERALIZED_TRIPLESTR_LIT`.

The construct `identifier"string literal"` (without whitespace between the identifier and the opening quotation mark) is a generalized raw string literal. It is a shortcut for the construct `identifier(r"string literal")`, so it denotes a procedure call with a raw string literal as its only argument. Generalized raw string literals are especially convenient for embedding mini languages directly into Nim (for example regular expressions).

The construct `identifier"""string literal"""` exists too. It is a shortcut for `identifier("""string literal""")`.

Escape sequence	Meaning
`\r`, `\c`	carriage return
`\n`, `\l`	line feed
`\f`	form feed
`\t`	tabulator
`\v`	vertical tabulator
`\\`	backslash
`\"`	quotation mark
`\'`	apostrophe
\ '0'..'9'+	character with decimal value d; all decimal digits directly following are used for the character
`\a`	alert
`\b`	backspace
`\e`	escape [ESC]
`\x` HH	character with hex value HH; exactly two hex digits are allowed

3.11 Character literals

Character literals are enclosed in single quotes '' and can contain the same escape sequences as strings - with one exception: the platform dependent newline (`\p`) is not allowed as it may be wider than one character (often it is the pair CR/LF for example). Here are the valid escape sequences for character literals:

A character is not an Unicode character but a single byte. The reason for this is efficiency: for the overwhelming majority of use-cases, the resulting programs will still handle UTF-8 properly as UTF-8 was specially designed for this. Another reason is that Nim can thus support `array[char, int]` or `set[char]` efficiently as many algorithms rely on this feature. The `Rune` type is used for Unicode characters, it can represent any Unicode character. `Rune` is declared in the unicode module.

3.12 Numerical constants

Numerical constants are of a single type and have the form:

```
hexdigit = digit | 'A'..'F' | 'a'..'f'
octdigit = '0'..'7'
bindigit = '0'..'1'
HEX_LIT = '0' ('x' | 'X' ) hexdigit ( ['_'] hexdigit )*
DEC_LIT = digit ( ['_'] digit )*
OCT_LIT = '0' 'o' octdigit ( ['_'] octdigit )*
BIN_LIT = '0' ('b' | 'B' ) bindigit ( ['_'] bindigit )*

INT_LIT = HEX_LIT
        | DEC_LIT
        | OCT_LIT
        | BIN_LIT

INT8_LIT = INT_LIT ['\''] ('i' | 'I') '8'
INT16_LIT = INT_LIT ['\''] ('i' | 'I') '16'
INT32_LIT = INT_LIT ['\''] ('i' | 'I') '32'
INT64_LIT = INT_LIT ['\''] ('i' | 'I') '64'

UINT_LIT = INT_LIT ['\''] ('u' | 'U')
UINT8_LIT = INT_LIT ['\''] ('u' | 'U') '8'
UINT16_LIT = INT_LIT ['\''] ('u' | 'U') '16'
UINT32_LIT = INT_LIT ['\''] ('u' | 'U') '32'
UINT64_LIT = INT_LIT ['\''] ('u' | 'U') '64'

exponent = ('e' | 'E' ) ['+' | '-'] digit ( ['_'] digit )*
FLOAT_LIT = digit (['_'] digit)* (('.' digit (['_'] digit)* [exponent]) |exponent)
FLOAT32_SUFFIX = ('f' | 'F') ['32']
FLOAT32_LIT = HEX_LIT '\'' FLOAT32_SUFFIX
```

Type Suffix	Resulting type of literal
`'i8`	int8
`'i16`	int16
`'i32`	int32
`'i64`	int64
`'u`	uint
`'u8`	uint8
`'u16`	uint16
`'u32`	uint32
`'u64`	uint64
`'f`	float32
`'d`	float64
`'f32`	float32
`'f64`	float64

```
            | (FLOAT_LIT | DEC_LIT | OCT_LIT | BIN_LIT) ['\''] FLOAT32_SUFFIX
FLOAT64_SUFFIX = ( ('f' | 'F') '64' ) | 'd' | 'D'
FLOAT64_LIT = HEX_LIT '\'' FLOAT64_SUFFIX
            | (FLOAT_LIT | DEC_LIT | OCT_LIT | BIN_LIT) ['\''] FLOAT64_SUFFIX
```

As can be seen in the productions, numerical constants can contain underscores for readability. Integer and floating point literals may be given in decimal (no prefix), binary (prefix `0b`), octal (prefix `0o`) and hexadecimal (prefix `0x`) notation.

There exists a literal for each numerical type that is defined. The suffix starting with an apostrophe (”’) is called a type suffix. Literals without a type suffix are of an integer type, unless the literal contains a dot or `E|e` in which case it is of type `float`. This integer type is `int` if the literal is in the range `low(i32)..high(i32)`, otherwise it is `int64`. For notational convenience the apostrophe of a type suffix is optional if it is not ambiguous (only hexadecimal floating point literals with a type suffix can be ambiguous).

The type suffixes are:

Floating point literals may also be in binary, octal or hexadecimal notation: `0B0_10001110100_000010100100011110` is approximately 1.72826e35 according to the IEEE floating point standard.

Literals are bounds checked so that they fit the datatype. Non base-10 literals are used mainly for flags and bit pattern representations, therefore bounds checking is done on bit width, not value range. If the literal fits in the bit width of the datatype, it is accepted. Hence: 0b10000000’u8 == 0x80’u8 == 128, but, 0b10000000’i8 == 0x80’i8 == -1 instead of causing an overflow error.

3.13 Operators

Nim allows user defined operators. An operator is any combination of the following characters:

```
=     +     -     *     /     <     >
@     $     ~     &     %     |
!     ?     ^     .     :     \
```

(The grammar uses the terminal OPR to refer to operator symbols as defined here.)

These keywords are also operators: `and or not xor shl shr div mod in notin is isnot of as from`.

. =, :, :: are not available as general operators; they are used for other notational purposes.

`*:` is as a special case treated as the two tokens * and : (to support `var v*: T`).

The `not` keyword is always a unary operator, `a not b` is parsed as `a(not b)`, not as `(a) not (b)`.

3.14 Other tokens

The following strings denote other tokens:

Precedence level	Operators	First character	Terminal symbol
10 (highest)		`$ ^`	OP10
9	`* / div mod shl shr %`	`* % \ /`	OP9
8	`+ -`	`+ - ~ |`	OP8
7	`&`	`&`	OP7
6	`..`	`.`	OP6
5	`== <= < >= > != in notin is isnot not of as from`	`= < > !`	OP5
4	`and`		OP4
3	`or xor`		OP3
2		`@ : ?`	OP2
1	*assignment operator* (like +=, *=)		OP1
0 (lowest)	*arrow like operator* (like ->, =>)		OP0

`` ` ( ) { } [ ] , ; [. .] {. .} (. .) [: ``

The slice operator `..` takes precedence over other tokens that contain a dot: `{..}` are the three tokens `{`, `..`, `}` and not the two tokens `{.`, `.}`.

4 Syntax

This section lists Nim's standard syntax. How the parser handles the indentation is already described in the Lexical Analysis3 section.

Nim allows user-definable operators. Binary operators have 11 different levels of precedence.

4.1 Associativity

Binary operators whose first character is `^` are right-associative, all other binary operators are left-associative.

```
proc `^/`(x, y: float): float =
  # a right-associative division operator
  result = x / y
echo 12 ^/ 4 ^/ 8 # 24.0 (4 / 8 = 0.5, then 12 / 0.5 = 24.0)
echo 12  / 4  / 8 # 0.375 (12 / 4 = 3.0, then 3 / 8 = 0.375)
```

4.2 Precedence

Unary operators always bind stronger than any binary operator: `$a + b` is `($a) + b` and not `$(a + b)`.

If an unary operator's first character is `@` it is a sigil-like operator which binds stronger than a `primarySuffix`: `@x.abc` is parsed as `(@x).abc` whereas `$x.abc` is parsed as `$(x.abc)`.

For binary operators that are not keywords the precedence is determined by the following rules:

Operators ending in either `->`, `~>` or `=>` are called arrow like, and have the lowest precedence of all operators.

If the operator ends with `=` and its first character is none of `<`, `>`, `!`, `=`, `~`, `?`, it is an *assignment operator* which has the second lowest precedence.

Otherwise precedence is determined by the first character.

Whether an operator is used a prefix operator is also affected by preceding whitespace (this parsing change was introduced with version 0.13.0):

```
echo $foo
# is parsed as
echo($foo)
```

Spacing also determines whether `(a, b)` is parsed as an the argument list of a call or whether it is parsed as a tuple constructor:

```
echo(1, 2) # pass 1 and 2 to echo

echo (1, 2) # pass the tuple (1, 2) to echo
```

4.3 Grammar

The grammar's start symbol is `module`.

```
# This file is generated by compiler/parser.nim.
module = stmt ^* (';' / IND{=})
comma = ',' COMMENT?
semicolon = ';' COMMENT?
colon = ':' COMMENT?
colcom = ':' COMMENT?
operator =  OP0 | OP1 | OP2 | OP3 | OP4 | OP5 | OP6 | OP7 | OP8 | OP9
         | 'or' | 'xor' | 'and'
         | 'is' | 'isnot' | 'in' | 'notin' | 'of' | 'as' | 'from' |
         | 'div' | 'mod' | 'shl' | 'shr' | 'not' | 'static' | '..'
prefixOperator = operator
optInd = COMMENT? IND?
optPar = (IND{>} | IND{=})?
simpleExpr = arrowExpr (OP0 optInd arrowExpr)* pragma?
arrowExpr = assignExpr (OP1 optInd assignExpr)*
assignExpr = orExpr (OP2 optInd orExpr)*
orExpr = andExpr (OP3 optInd andExpr)*
andExpr = cmpExpr (OP4 optInd cmpExpr)*
cmpExpr = sliceExpr (OP5 optInd sliceExpr)*
sliceExpr = ampExpr (OP6 optInd ampExpr)*
ampExpr = plusExpr (OP7 optInd plusExpr)*
plusExpr = mulExpr (OP8 optInd mulExpr)*
mulExpr = dollarExpr (OP9 optInd dollarExpr)*
dollarExpr = primary (OP10 optInd primary)*
symbol = '`' (KEYW|IDENT|literal|(operator|'('|')'|'['|']'|'{'|'}'|'=')+)+ '`'
       | IDENT | KEYW
exprColonEqExpr = expr (':'|'=' expr)?
exprList = expr ^+ comma
exprColonEqExprList = exprColonEqExpr (comma exprColonEqExpr)* (comma)?
dotExpr = expr '.' optInd (symbol | '[:' exprList ']')
explicitGenericInstantiation = '[:' exprList ']' ( '(' exprColonEqExpr ')' )?
qualifiedIdent = symbol ('.' optInd symbol)?
setOrTableConstr = '{' ((exprColonEqExpr comma)* | ':' ) '}'
castExpr = 'cast' '[' optInd typeDesc optPar ']' '(' optInd expr optPar ')'
parKeyw = 'discard' | 'include' | 'if' | 'while' | 'case' | 'try'
        | 'finally' | 'except' | 'for' | 'block' | 'const' | 'let'
        | 'when' | 'var' | 'mixin'
par = '(' optInd
          ( &parKeyw complexOrSimpleStmt ^+ ';'
          | ';' complexOrSimpleStmt ^+ ';'
          | pragmaStmt
          | simpleExpr ( ('=' expr (';' complexOrSimpleStmt ^+ ';' )? )
                       | (':' expr (',' exprColonEqExpr     ^+ ',' )? ) ) )
          optPar ')'
literal = | INT_LIT | INT8_LIT | INT16_LIT | INT32_LIT | INT64_LIT
          | UINT_LIT | UINT8_LIT | UINT16_LIT | UINT32_LIT | UINT64_LIT
          | FLOAT_LIT | FLOAT32_LIT | FLOAT64_LIT
          | STR_LIT | RSTR_LIT | TRIPLESTR_LIT
          | CHAR_LIT
          | NIL
generalizedLit = GENERALIZED_STR_LIT | GENERALIZED_TRIPLESTR_LIT
identOrLiteral = generalizedLit | symbol | literal
               | par | arrayConstr | setOrTableConstr
               | castExpr
```

```
tupleConstr = '(' optInd (exprColonEqExpr comma?)* optPar ')'
arrayConstr = '[' optInd (exprColonEqExpr comma?)* optPar ']'
primarySuffix = '(' (exprColonEqExpr comma?)* ')'
      | '.' optInd symbol generalizedLit?
      | '[' optInd exprColonEqExprList optPar ']'
      | '{' optInd exprColonEqExprList optPar '}'
      | &( '`'|IDENT|literal|'cast'|'addr'|'type') expr # command syntax
condExpr = expr colcom expr optInd
        ('elif' expr colcom expr optInd)*
         'else' colcom expr
ifExpr = 'if' condExpr
whenExpr = 'when' condExpr
pragma = '{.' optInd (exprColonEqExpr comma?)* optPar ('.}' | '}')
identVis = symbol OPR?  # postfix position
identVisDot = symbol '.' optInd symbol OPR?
identWithPragma = identVis pragma?
identWithPragmaDot = identVisDot pragma?
declColonEquals = identWithPragma (comma identWithPragma)* comma?
                  (':' optInd typeDesc)? ('=' optInd expr)?
identColonEquals = IDENT (comma IDENT)* comma?
     (':' optInd typeDesc)? ('=' optInd expr)?)
inlTupleDecl = 'tuple'
    '[' optInd  (identColonEquals (comma/semicolon)?)*  optPar ']'
extTupleDecl = 'tuple'
    COMMENT? (IND{>} identColonEquals (IND{=} identColonEquals)*)?
tupleClass = 'tuple'
paramList = '(' declColonEquals ^* (comma/semicolon) ')'
paramListArrow = paramList? ('->' optInd typeDesc)?
paramListColon = paramList? (':' optInd typeDesc)?
doBlock = 'do' paramListArrow pragma? colcom stmt
procExpr = 'proc' paramListColon pragma? ('=' COMMENT? stmt)?
distinct = 'distinct' optInd typeDesc
forStmt = 'for' (identWithPragma ^+ comma) 'in' expr colcom stmt
forExpr = forStmt
expr = (blockExpr
      | ifExpr
      | whenExpr
      | caseStmt
      | forExpr
      | tryExpr)
      / simpleExpr
typeKeyw = 'var' | 'out' | 'ref' | 'ptr' | 'shared' | 'tuple'
         | 'proc' | 'iterator' | 'distinct' | 'object' | 'enum'
primary = typeKeyw optInd typeDesc
        /  prefixOperator* identOrLiteral primarySuffix*
        / 'bind' primary
typeDesc = simpleExpr ('not' expr)?
typeDefAux = simpleExpr ('not' expr)?
           | 'concept' typeClass
postExprBlocks = ':' stmt? ( IND{=} doBlock
                           | IND{=} 'of' exprList ':' stmt
                           | IND{=} 'elif' expr ':' stmt
                           | IND{=} 'except' exprList ':' stmt
                           | IND{=} 'else' ':' stmt )*
exprStmt = simpleExpr
         (( '=' optInd expr colonBody? )
         / ( expr ^+ comma
             postExprBlocks
           ))?
importStmt = 'import' optInd expr
              ((comma expr)*
              / 'except' optInd (expr ^+ comma))
exportStmt = 'export' optInd expr
              ((comma expr)*
              / 'except' optInd (expr ^+ comma))
includeStmt = 'include' optInd expr ^+ comma
fromStmt = 'from' expr 'import' optInd expr (comma expr)*
returnStmt = 'return' optInd expr?
raiseStmt = 'raise' optInd expr?
yieldStmt = 'yield' optInd expr?
```

```
discardStmt = 'discard' optInd expr?
breakStmt = 'break' optInd expr?
continueStmt = 'break' optInd expr?
condStmt = expr colcom stmt COMMENT?
           (IND{=} 'elif' expr colcom stmt)*
           (IND{=} 'else' colcom stmt)?
ifStmt = 'if' condStmt
whenStmt = 'when' condStmt
whileStmt = 'while' expr colcom stmt
ofBranch = 'of' exprList colcom stmt
ofBranches = ofBranch (IND{=} ofBranch)*
                      (IND{=} 'elif' expr colcom stmt)*
                      (IND{=} 'else' colcom stmt)?
caseStmt = 'case' expr ':'? COMMENT?
            (IND{>} ofBranches DED
            | IND{=} ofBranches)
tryStmt = 'try' colcom stmt &(IND{=}? 'except'|'finally')
           (IND{=}? 'except' exprList colcom stmt)*
           (IND{=}? 'finally' colcom stmt)?
tryExpr = 'try' colcom stmt &(optInd 'except'|'finally')
           (optInd 'except' exprList colcom stmt)*
           (optInd 'finally' colcom stmt)?
exceptBlock = 'except' colcom stmt
blockStmt = 'block' symbol? colcom stmt
blockExpr = 'block' symbol? colcom stmt
staticStmt = 'static' colcom stmt
deferStmt = 'defer' colcom stmt
asmStmt = 'asm' pragma? (STR_LIT | RSTR_LIT | TRIPLESTR_LIT)
genericParam = symbol (comma symbol)* (colon expr)? ('=' optInd expr)?
genericParamList = '[' optInd
  genericParam ^* (comma/semicolon) optPar ']'
pattern = '{' stmt '}'
indAndComment = (IND{>} COMMENT)? | COMMENT?
routine = optInd identVis pattern? genericParamList?
  paramListColon pragma? ('=' COMMENT? stmt)? indAndComment
commentStmt = COMMENT
section(RULE) = COMMENT? RULE / (IND{>} (RULE / COMMENT)^+IND{=} DED)
enum = 'enum' optInd (symbol pragma? optInd ('=' optInd expr COMMENT?)? comma?)+
objectWhen = 'when' expr colcom objectPart COMMENT?
            ('elif' expr colcom objectPart COMMENT?)*
            ('else' colcom objectPart COMMENT?)?
objectBranch = 'of' exprList colcom objectPart
objectBranches = objectBranch (IND{=} objectBranch)*
                      (IND{=} 'elif' expr colcom objectPart)*
                      (IND{=} 'else' colcom objectPart)?
objectCase = 'case' identWithPragma ':' typeDesc ':'? COMMENT?
            (IND{>} objectBranches DED
            | IND{=} objectBranches)
objectPart = IND{>} objectPart^+IND{=} DED
           / objectWhen / objectCase / 'nil' / 'discard' / declColonEquals
object = 'object' pragma? ('of' typeDesc)? COMMENT? objectPart
typeClassParam = ('var' | 'out')? symbol
typeClass = typeClassParam ^* ',' (pragma)? ('of' typeDesc ^* ',')?
              &IND{>} stmt
typeDef = identWithPragmaDot genericParamList? '=' optInd typeDefAux
            indAndComment? / identVisDot genericParamList? pragma '=' optInd typeDefAux
            indAndComment?
varTuple = '(' optInd identWithPragma ^+ comma optPar ')' '=' optInd expr
colonBody = colcom stmt postExprBlocks?
variable = (varTuple / identColonEquals) colonBody? indAndComment
constant = (varTuple / identWithPragma) (colon typeDesc)? '=' optInd expr indAndComment
bindStmt = 'bind' optInd qualifiedIdent ^+ comma
mixinStmt = 'mixin' optInd qualifiedIdent ^+ comma
pragmaStmt = pragma (':' COMMENT? stmt)?
simpleStmt = ((returnStmt | raiseStmt | yieldStmt | discardStmt | breakStmt
           | continueStmt | pragmaStmt | importStmt | exportStmt | fromStmt
           | includeStmt | commentStmt) / exprStmt) COMMENT?
complexOrSimpleStmt = (ifStmt | whenStmt | whileStmt
                    | tryStmt | forStmt
                    | blockStmt | staticStmt | deferStmt | asmStmt
```

```
                    | 'proc' routine
                    | 'func' routine
                    | 'method' routine
                    | 'iterator' routine
                    | 'macro' routine
                    | 'template' routine
                    | 'converter' routine
                    | 'type' section(typeDef)
                    | 'const' section(constant)
                    | ('let' | 'var' | 'using') section(variable)
                    | bindStmt | mixinStmt)
                    / simpleStmt
stmt = (IND{>} complexOrSimpleStmt^+(IND{=} / ';') DED)
     / simpleStmt ^+ ';'
```

5 Order of evaluation

Order of evaluation is strictly left-to-right, inside-out as it is typical for most others imperative programming languages:

```
var s = ""

proc p(arg: int): int =
  s.add $arg
  result = arg

discard p(p(1) + p(2))

doAssert s == "123"
```

Assignments are not special, the left-hand-side expression is evaluated before the right-hand side:

```
var v = 0
proc getI(): int =
  result = v
  inc v

var a, b: array[0..2, int]

proc someCopy(a: var int; b: int) = a = b

a[getI()] = getI()

doAssert a == [1, 0, 0]

v = 0
someCopy(b[getI()], getI())

doAssert b == [1, 0, 0]
```

Rationale: Consistency with overloaded assignment or assignment-like operations, `a = b` can be read as `performSomeCopy(a, b)`.

However, the concept of "order of evaluation" is only applicable after the code was normalized: The normalization involves template expansions and argument reorderings that have been passed to named parameters:

```
var s = ""

proc p(): int =
  s.add "p"
  result = 5

proc q(): int =
  s.add "q"
  result = 3
```

```
# Evaluation order is 'b' before 'a' due to template
# expansion's semantics.
template swapArgs(a, b): untyped =
  b + a

doAssert swapArgs(p() + q(), q() - p()) == 6
doAssert s == "qppq"

# Evaluation order is not influenced by named parameters:
proc construct(first, second: int) =
  discard

# 'p' is evaluated before 'q'!
construct(second = q(), first = p())

doAssert s == "qppqpq"
```

Rationale: This is far easier to implement than hypothetical alternatives.

6 Constants and Constant Expressions

A constant is a symbol that is bound to the value of a constant expression. Constant expressions are restricted to depend only on the following categories of values and operations, because these are either built into the language or declared and evaluated before semantic analysis of the constant expression:

- literals
- built-in operators
- previously declared constants and compile-time variables
- previously declared macros and templates
- previously declared procedures that have no side effects beyond possibly modifying compile-time variables

A constant expression can contain code blocks that may internally use all Nim features supported at compile time (as detailed in the next section below). Within such a code block, it is possible to declare variables and then later read and update them, or declare variables and pass them to procedures that modify them. However, the code in such a block must still adhere to the restrictions listed above for referencing values and operations outside the block.

The ability to access and modify compile-time variables adds flexibility to constant expressions that may be surprising to those coming from other statically typed languages. For example, the following code echoes the beginning of the Fibonacci series **at compile time**. (This is a demonstration of flexibility in defining constants, not a recommended style for solving this problem!)

```
import strformat

var fib_n {.compileTime.}: int
var fib_prev {.compileTime.}: int
var fib_prev_prev {.compileTime.}: int

proc next_fib(): int =
  result = if fib_n < 2:
    fib_n
  else:
    fib_prev_prev + fib_prev
  inc(fib_n)
  fib_prev_prev = fib_prev
  fib_prev = result

const f0 = next_fib()
const f1 = next_fib()

const display_fib = block:
```

```
  const f2 = next_fib()
  var result = fmt"Fibonacci sequence: {f0}, {f1}, {f2}"
  for i in 3..12:
    add(result, fmt", {next_fib()}")
  result

static:
  echo display_fib
```

7 Restrictions on Compile-Time Execution

Nim code that will be executed at compile time cannot use the following language features:

- methods
- closure iterators
- the `cast` operator
- reference (pointer) types
- FFI

The use of wrappers that use FFI and/or `cast` is also disallowed. Note that these wrappers include the ones in the standard libraries.

Some or all of these restrictions are likely to be lifted over time.

8 Types

All expressions have a type which is known during semantic analysis. Nim is statically typed. One can declare new types, which is in essence defining an identifier that can be used to denote this custom type.

These are the major type classes:

- ordinal types (consist of integer, bool, character, enumeration (and subranges thereof) types)
- floating point types
- string type
- structured types
- reference (pointer) type
- procedural type
- generic type

8.1 Ordinal types

Ordinal types have the following characteristics:

- Ordinal types are countable and ordered. This property allows the operation of functions as `inc`, `ord`, `dec` on ordinal types to be defined.
- Ordinal values have a smallest possible value. Trying to count further down than the smallest value produces a panic or a static error.
- Ordinal values have a largest possible value. Trying to count further than the largest value produces a panic or a static error.

Integers, bool, characters and enumeration types (and subranges of these types) belong to ordinal types. For reasons of simplicity of implementation the types `uint` and `uint64` are not ordinal types. (This will be changed in later versions of the language.)

A distinct type is an ordinal type if its base type is an ordinal type.

operation	meaning
`a +% b`	unsigned integer addition
`a -% b`	unsigned integer subtraction
`a *% b`	unsigned integer multiplication
`a /% b`	unsigned integer division
`a %% b`	unsigned integer modulo operation
`a <% b`	treat `a` and `b` as unsigned and compare
`a <=% b`	treat `a` and `b` as unsigned and compare
`ze(a)`	extends the bits of `a` with zeros until it has the width of the `int` type
`toU8(a)`	treats `a` as unsigned and converts it to an unsigned integer of 8 bits (but still the `int8` type)
`toU16(a)`	treats `a` as unsigned and converts it to an unsigned integer of 16 bits (but still the `int16` type)
`toU32(a)`	treats `a` as unsigned and converts it to an unsigned integer of 32 bits (but still the `int32` type)

8.2 Pre-defined integer types

These integer types are pre-defined:

`int` the generic signed integer type; its size is platform dependent and has the same size as a pointer. This type should be used in general. An integer literal that has no type suffix is of this type if it is in the range `low(int32)..high(int32)` otherwise the literal's type is `int64`.

intXX additional signed integer types of XX bits use this naming scheme (example: int16 is a 16 bit wide integer). The current implementation supports `int8`, `int16`, `int32`, `int64`. Literals of these types have the suffix 'iXX.

`uint` the generic unsigned integer type; its size is platform dependent and has the same size as a pointer. An integer literal with the type suffix `'u` is of this type.

uintXX additional unsigned integer types of XX bits use this naming scheme (example: uint16 is a 16 bit wide unsigned integer). The current implementation supports `uint8`, `uint16`, `uint32`, `uint64`. Literals of these types have the suffix 'uXX. Unsigned operations all wrap around; they cannot lead to over- or underflow errors.

In addition to the usual arithmetic operators for signed and unsigned integers (`+ - *` etc.) there are also operators that formally work on *signed* integers but treat their arguments as *unsigned*: They are mostly provided for backwards compatibility with older versions of the language that lacked unsigned integer types. These unsigned operations for signed integers use the `%` suffix as convention:

Automatic type conversion is performed in expressions where different kinds of integer types are used: the smaller type is converted to the larger.

A narrowing type conversion converts a larger to a smaller type (for example `int32 -> int16`. A widening type conversion converts a smaller type to a larger type (for example `int16 -> int32`). In Nim only widening type conversions are *implicit*:

```
var myInt16 = 5i16
var myInt: int
myInt16 + 34     # of type ''int16''
myInt16 + myInt  # of type ''int''
myInt16 + 2i32   # of type ''int32''
```

However, `int` literals are implicitly convertible to a smaller integer type if the literal's value fits this smaller type and such a conversion is less expensive than other implicit conversions, so `myInt16 + 34` produces an `int16` result.

For further details, see Convertible relation.

8.3 Subrange types

A subrange type is a range of values from an ordinal or floating point type (the base type). To define a subrange type, one must specify its limiting values – the lowest and highest value of the type. For example:

```
type
  Subrange = range[0..5]
  PositiveFloat = range[0.0..Inf]
```

`Subrange` is a subrange of an integer which can only hold the values 0 to 5. `PositiveFloat` defines a subrange of all positive floating point values. NaN does not belong to any subrange of floating point types. Assigning any other value to a variable of type `Subrange` is a panic (or a static error if it can be determined during semantic analysis). Assignments from the base type to one of its subrange types (and vice versa) are allowed.

A subrange type has the same size as its base type (`int` in the Subrange example).

8.4 Pre-defined floating point types

The following floating point types are pre-defined:

`float` the generic floating point type; its size used to be platform dependent, but now it is always mapped to `float64`. This type should be used in general.

floatXX an implementation may define additional floating point types of XX bits using this naming scheme (example: float64 is a 64 bit wide float). The current implementation supports `float32` and `float64`. Literals of these types have the suffix 'fXX.

Automatic type conversion in expressions with different kinds of floating point types is performed: See Convertible relation for further details. Arithmetic performed on floating point types follows the IEEE standard. Integer types are not converted to floating point types automatically and vice versa.

The IEEE standard defines five types of floating-point exceptions:

- Invalid: operations with mathematically invalid operands, for example 0.0/0.0, sqrt(-1.0), and log(-37.8).
- Division by zero: divisor is zero and dividend is a finite nonzero number, for example 1.0/0.0.
- Overflow: operation produces a result that exceeds the range of the exponent, for example MAXDOUBLE+0.0000000000001e308.
- Underflow: operation produces a result that is too small to be represented as a normal number, for example, MINDOUBLE * MINDOUBLE.
- Inexact: operation produces a result that cannot be represented with infinite precision, for example, 2.0 / 3.0, log(1.1) and 0.1 in input.

The IEEE exceptions are either ignored during execution or mapped to the Nim exceptions: FloatInvalidOpDefect, FloatDivByZeroDefect, FloatOverflowDefect, FloatUnderflowDefect, and FloatInexactDefect. These exceptions inherit from the FloatingPointDefect base class.

Nim provides the pragmas nanChecks and infChecks to control whether the IEEE exceptions are ignored or trap a Nim exception:

```
{.nanChecks: on, infChecks: on.}
var a = 1.0
var b = 0.0
echo b / b # raises FloatInvalidOpDefect
echo a / b # raises FloatOverflowDefect
```

In the current implementation `FloatDivByZeroDefect` and `FloatInexactDefect` are never raised. `FloatOverflowDefect` is raised instead of `FloatDivByZeroDefect`. There is also a floatChecks pragma that is a short-cut for the combination of `nanChecks` and `infChecks` pragmas. `floatChecks` are turned off as default.

The only operations that are affected by the `floatChecks` pragma are the `+`, `-`, `*`, `/` operators for floating point types.

An implementation should always use the maximum precision available to evaluate floating pointer values during semantic analysis; this means expressions like `0.09'f32 + 0.01'f32 == 0.09'f64 + 0.01'f64` that are evaluating during constant folding are true.

8.5 Boolean type

The boolean type is named bool in Nim and can be one of the two pre-defined values `true` and `false`. Conditions in `while`, `if`, `elif`, `when`-statements need to be of type `bool`.

This condition holds:

```
ord(false) == 0 and ord(true) == 1
```

The operators `not, and, or, xor, <, <=, >, >=, !=, ==` are defined for the bool type. The `and` and `or` operators perform short-cut evaluation. Example:

```
while p != nil and p.name != "xyz":
  # p.name is not evaluated if p == nil
  p = p.next
```

The size of the bool type is one byte.

8.6 Character type

The character type is named `char` in Nim. Its size is one byte. Thus it cannot represent an UTF-8 character, but a part of it. The reason for this is efficiency: for the overwhelming majority of use-cases, the resulting programs will still handle UTF-8 properly as UTF-8 was specially designed for this. Another reason is that Nim can support `array[char, int]` or `set[char]` efficiently as many algorithms rely on this feature. The `Rune` type is used for Unicode characters, it can represent any Unicode character. `Rune` is declared in the unicode module.

8.7 Enumeration types

Enumeration types define a new type whose values consist of the ones specified. The values are ordered. Example:

```
type
  Direction = enum
    north, east, south, west
```

Now the following holds:

```
ord(north) == 0
ord(east) == 1
ord(south) == 2
ord(west) == 3

# Also allowed:
ord(Direction.west) == 3
```

Thus, north < east < south < west. The comparison operators can be used with enumeration types. Instead of `north` etc, the enum value can also be qualified with the enum type that it resides in, `Direction.north`.

For better interfacing to other programming languages, the fields of enum types can be assigned an explicit ordinal value. However, the ordinal values have to be in ascending order. A field whose ordinal value is not explicitly given is assigned the value of the previous field + 1.

An explicit ordered enum can have *holes*:

```nim
type
  TokenType = enum
    a = 2, b = 4, c = 89 # holes are valid
```

However, it is then not an ordinal anymore, so it is not possible to use these enums as an index type for arrays. The procedures `inc`, `dec`, `succ` and `pred` are not available for them either.

The compiler supports the built-in stringify operator `$` for enumerations. The stringify's result can be controlled by explicitly giving the string values to use:

```nim
type
  MyEnum = enum
    valueA = (0, "my value A"),
    valueB = "value B",
    valueC = 2,
    valueD = (3, "abc")
```

As can be seen from the example, it is possible to both specify a field's ordinal value and its string value by using a tuple. It is also possible to only specify one of them.

An enum can be marked with the `pure` pragma so that it's fields are added to a special module specific hidden scope that is only queried as the last attempt. Only non-ambiguous symbols are added to this scope. But one can always access these via type qualification written as `MyEnum.value`:

```nim
type
  MyEnum {.pure.} = enum
    valueA, valueB, valueC, valueD, amb

  OtherEnum {.pure.} = enum
    valueX, valueY, valueZ, amb


echo valueA # MyEnum.valueA
echo amb    # Error: Unclear whether it's MyEnum.amb or OtherEnum.amb
echo MyEnum.amb # OK.
```

To implement bit fields with enums see Bit fields

8.8 String type

All string literals are of the type `string`. A string in Nim is very similar to a sequence of characters. However, strings in Nim are both zero-terminated and have a length field. One can retrieve the length with the builtin `len` procedure; the length never counts the terminating zero.

The terminating zero cannot be accessed unless the string is converted to the `cstring` type first. The terminating zero assures that this conversion can be done in O(1) and without any allocations.

The assignment operator for strings always copies the string. The `&` operator concatenates strings.

Most native Nim types support conversion to strings with the special `$` proc. When calling the `echo` proc, for example, the built-in stringify operation for the parameter is called:

```nim
echo 3 # calls `$` for `int`
```

Whenever a user creates a specialized object, implementation of this procedure provides for `string` representation.

```nim
type
  Person = object
    name: string
    age: int

proc `$`(p: Person): string = # `$` always returns a string
  result = p.name & " is " &
          $p.age & # we *need* the `$` in front of p.age which
                   # is natively an integer to convert it to
                   # a string
          " years old."
```

While `$p.name` can also be used, the `$` operation on a string does nothing. Note that we cannot rely on automatic conversion from an `int` to a `string` like we can for the `echo` proc.

Strings are compared by their lexicographical order. All comparison operators are available. Strings can be indexed like arrays (lower bound is 0). Unlike arrays, they can be used in case statements:

```
case paramStr(i)
of "-v": incl(options, optVerbose)
of "-h", "-?": incl(options, optHelp)
else: write(stdout, "invalid command line option!\n")
```

Per convention, all strings are UTF-8 strings, but this is not enforced. For example, when reading strings from binary files, they are merely a sequence of bytes. The index operation `s[i]` means the i-th *char* of `s`, not the i-th *unichar*. The iterator `runes` from the unicode module can be used for iteration over all Unicode characters.

8.9 cstring type

The `cstring` type meaning `compatible string` is the native representation of a string for the compilation backend. For the C backend the `cstring` type represents a pointer to a zero-terminated char array compatible to the type `char*` in Ansi C. Its primary purpose lies in easy interfacing with C. The index operation `s[i]` means the i-th *char* of `s`; however no bounds checking for `cstring` is performed making the index operation unsafe.

A Nim `string` is implicitly convertible to `cstring` for convenience. If a Nim string is passed to a C-style variadic proc, it is implicitly converted to `cstring` too:

```
proc printf(formatstr: cstring) {.importc: "printf", varargs,
                                  header: "<stdio.h>".}

printf("This works %s", "as expected")
```

Even though the conversion is implicit, it is not *safe*: The garbage collector does not consider a `cstring` to be a root and may collect the underlying memory. However in practice this almost never happens as the GC considers stack roots conservatively. One can use the builtin procs `GC_ref` and `GC_unref` to keep the string data alive for the rare cases where it does not work.

A `$` proc is defined for cstrings that returns a string. Thus to get a nim string from a cstring:

```
var str: string = "Hello!"
var cstr: cstring = str
var newstr: string = $cstr
```

8.10 Structured types

A variable of a structured type can hold multiple values at the same time. Structured types can be nested to unlimited levels. Arrays, sequences, tuples, objects and sets belong to the structured types.

8.11 Array and sequence types

Arrays are a homogeneous type, meaning that each element in the array has the same type. Arrays always have a fixed length specified as a constant expression (except for open arrays). They can be indexed by any ordinal type. A parameter `A` may be an *open array*, in which case it is indexed by integers from 0 to `len(A)-1`. An array expression may be constructed by the array constructor `[]`. The element type of this array expression is inferred from the type of the first element. All other elements need to be implicitly convertible to this type.

An array type can be defined using the `array[size, T]` syntax, or using `array[lo..hi, T]` for arrays that start at an index other than zero.

Sequences are similar to arrays but of dynamic length which may change during runtime (like strings). Sequences are implemented as growable arrays, allocating pieces of memory as items are added. A sequence `S` is always indexed by integers from 0 to `len(S)-1` and its bounds are checked. Sequences can be constructed by the array constructor `[]` in conjunction with the array to sequence operator `@`. Another way to allocate space for a sequence is to call the built-in `newSeq` procedure.

A sequence may be passed to a parameter that is of type *open array*.

Example:

```
type
  IntArray = array[0..5, int] # an array that is indexed with 0..5
  IntSeq = seq[int] # a sequence of integers
var
  x: IntArray
  y: IntSeq
x = [1, 2, 3, 4, 5, 6]  # [] is the array constructor
y = @[1, 2, 3, 4, 5, 6] # the @ turns the array into a sequence

let z = [1.0, 2, 3, 4] # the type of z is array[0..3, float]
```

The lower bound of an array or sequence may be received by the built-in proc `low()`, the higher bound by `high()`. The length may be received by `len()`. `low()` for a sequence or an open array always returns 0, as this is the first valid index. One can append elements to a sequence with the `add()` proc or the `&` operator, and remove (and get) the last element of a sequence with the `pop()` proc.

The notation `x[i]` can be used to access the i-th element of `x`.

Arrays are always bounds checked (statically or at runtime). These checks can be disabled via pragmas or invoking the compiler with the `-boundChecks:off` command line switch.

An array constructor can have explicit indexes for readability:

```
type
  Values = enum
    valA, valB, valC

const
  lookupTable = [
    valA: "A",
    valB: "B",
    valC: "C"
  ]
```

If an index is left out, `succ(lastIndex)` is used as the index value:

```
type
  Values = enum
    valA, valB, valC, valD, valE

const
  lookupTable = [
    valA: "A",
    "B",
    valC: "C",
    "D", "e"
  ]
```

8.12 Open arrays

Often fixed size arrays turn out to be too inflexible; procedures should be able to deal with arrays of different sizes. The openarray type allows this; it can only be used for parameters. Openarrays are always indexed with an `int` starting at position 0. The `len`, `low` and `high` operations are available for open arrays too. Any array with a compatible base type can be passed to an openarray parameter, the index type does not matter. In addition to arrays sequences can also be passed to an open array parameter.

The openarray type cannot be nested: multidimensional openarrays are not supported because this is seldom needed and cannot be done efficiently.

```
proc testOpenArray(x: openArray[int]) = echo repr(x)

testOpenArray([1,2,3])  # array[]
testOpenArray(@[1,2,3]) # seq[]
```

8.13 Varargs

A `varargs` parameter is an openarray parameter that additionally allows to pass a variable number of arguments to a procedure. The compiler converts the list of arguments to an array implicitly:

```
proc myWriteln(f: File, a: varargs[string]) =
  for s in items(a):
    write(f, s)
  write(f, "\n")

myWriteln(stdout, "abc", "def", "xyz")
# is transformed to:
myWriteln(stdout, ["abc", "def", "xyz"])
```

This transformation is only done if the varargs parameter is the last parameter in the procedure header. It is also possible to perform type conversions in this context:

```
proc myWriteln(f: File, a: varargs[string, `$`]) =
  for s in items(a):
    write(f, s)
  write(f, "\n")

myWriteln(stdout, 123, "abc", 4.0)
# is transformed to:
myWriteln(stdout, [$123, $"def", $4.0])
```

In this example `$` is applied to any argument that is passed to the parameter `a`. (Note that `$` applied to strings is a nop.)

Note that an explicit array constructor passed to a `varargs` parameter is not wrapped in another implicit array construction:

```
proc takeV[T](a: varargs[T]) = discard

takeV([123, 2, 1]) # takeV's T is "int", not "array of int"
```

`varargs[typed]` is treated specially: It matches a variable list of arguments of arbitrary type but *always* constructs an implicit array. This is required so that the builtin `echo` proc does what is expected:

```
proc echo*(x: varargs[typed, `$`]) {...}

echo @[1, 2, 3]
# prints "@[1, 2, 3]" and not "123"
```

8.14 Unchecked arrays

The `UncheckedArray[T]` type is a special kind of `array` where its bounds are not checked. This is often useful to implement customized flexibly sized arrays. Additionally an unchecked array is translated into a C array of undetermined size:

```
type
  MySeq = object
    len, cap: int
    data: UncheckedArray[int]
```

Produces roughly this C code:

```
typedef struct {
  NI len;
  NI cap;
  NI data[];
} MySeq;
```

The base type of the unchecked array may not contain any GC'ed memory but this is currently not checked.

Future directions: GC'ed memory should be allowed in unchecked arrays and there should be an explicit annotation of how the GC is to determine the runtime size of the array.

8.15 Tuples and object types

A variable of a tuple or object type is a heterogeneous storage container. A tuple or object defines various named *fields* of a type. A tuple also defines a lexicographic *order* of the fields. Tuples are meant to be heterogeneous storage types with few abstractions. The `()` syntax can be used to construct tuples. The order of the fields in the constructor must match the order of the tuple's definition. Different tuple-types are *equivalent* if they specify the same fields of the same type in the same order. The *names* of the fields also have to be identical.

The assignment operator for tuples copies each component. The default assignment operator for objects copies each component. Overloading of the assignment operator is described here.

```
type
  Person = tuple[name: string, age: int] # type representing a person:
                                         # a person consists of a name
                                         # and an age
var
  person: Person
person = (name: "Peter", age: 30)
echo person.name
# the same, but less readable:
person = ("Peter", 30)
echo person[0]
```

A tuple with one unnamed field can be constructed with the parentheses and a trailing comma:

```
proc echoUnaryTuple(a: (int,)) =
  echo a[0]

echoUnaryTuple (1,)
```

In fact, a trailing comma is allowed for every tuple construction.

The implementation aligns the fields for best access performance. The alignment is compatible with the way the C compiler does it.

For consistency with `object` declarations, tuples in a `type` section can also be defined with indentation instead of `[]`:

```
type
  Person = tuple   # type representing a person
    name: string   # a person consists of a name
    age: Natural   # and an age
```

Objects provide many features that tuples do not. Object provide inheritance and the ability to hide fields from other modules. Objects with inheritance enabled have information about their type at runtime, so that the `of` operator can be used to determine the object's type. The `of` operator is similar to the `instanceof` operator in Java.

```
type
  Person = object of RootObj
    name*: string   # the * means that `name` is accessible from other modules
    age: int        # no * means that the field is hidden

  Student = ref object of Person # a student is a person
    id: int                      # with an id field

var
  student: Student
  person: Person
assert(student of Student) # is true
assert(student of Person) # also true
```

Object fields that should be visible from outside the defining module, have to be marked by `*`. In contrast to tuples, different object types are never *equivalent*, they are nominal types whereas tuples are structural. Objects that have no ancestor are implicitly `final` and thus have no hidden type information. One can use the `inheritable` pragma to introduce new object roots apart from `system.RootObj`.

```
type
  Person = object # example of a final object
    name*: string
    age: int

  Student = ref object of Person # Error: inheritance only works with non-final objects
    id: int
```

8.16 Object construction

Objects can also be created with an object construction expression that has the syntax `T(fieldA: valueA, fieldB: valueB, ...)` where `T` is an `object` type or a `ref object` type:

```
var student = Student(name: "Anton", age: 5, id: 3)
```

Note that, unlike tuples, objects require the field names along with their values. For a `ref object` type `system.new` is invoked implicitly.

8.17 Object variants

Often an object hierarchy is overkill in certain situations where simple variant types are needed. Object variants are tagged unions discriminated via a enumerated type used for runtime type flexibility, mirroring the concepts of *sum types* and *algebraic data types (ADTs)* as found in other languages.

An example:

```
# This is an example how an abstract syntax tree could be modelled in Nim
type
  NodeKind = enum  # the different node types
    nkInt,          # a leaf with an integer value
    nkFloat,        # a leaf with a float value
    nkString,       # a leaf with a string value
    nkAdd,          # an addition
    nkSub,          # a subtraction
    nkIf            # an if statement
  Node = ref NodeObj
  NodeObj = object
    case kind: NodeKind  # the ``kind`` field is the discriminator
    of nkInt: intVal: int
    of nkFloat: floatVal: float
    of nkString: strVal: string
    of nkAdd, nkSub:
      leftOp, rightOp: Node
    of nkIf:
      condition, thenPart, elsePart: Node

# create a new case object:
var n = Node(kind: nkIf, condition: nil)
# accessing n.thenPart is valid because the ``nkIf`` branch is active:
n.thenPart = Node(kind: nkFloat, floatVal: 2.0)

# the following statement raises an `FieldDefect` exception, because
# n.kind's value does not fit and the ``nkString`` branch is not active:
n.strVal = ""

# invalid: would change the active object branch:
n.kind = nkInt

var x = Node(kind: nkAdd, leftOp: Node(kind: nkInt, intVal: 4),
                          rightOp: Node(kind: nkInt, intVal: 2))
# valid: does not change the active object branch:
x.kind = nkSub
```

As can been seen from the example, an advantage to an object hierarchy is that no casting between different object types is needed. Yet, access to invalid object fields raises an exception.

The syntax of `case` in an object declaration follows closely the syntax of the `case` statement: The branches in a `case` section may be indented too.

In the example the `kind` field is called the discriminator: For safety its address cannot be taken and assignments to it are restricted: The new value must not lead to a change of the active object branch. Also, when the fields of a particular branch are specified during object construction, the corresponding discriminator value must be specified as a constant expression.

Instead of changing the active object branch, replace the old object in memory with a new one completely:

```
var x = Node(kind: nkAdd, leftOp: Node(kind: nkInt, intVal: 4),
                          rightOp: Node(kind: nkInt, intVal: 2))
# change the node's contents:
x[] = NodeObj(kind: nkString, strVal: "abc")
```

Starting with version 0.20 `system.reset` cannot be used anymore to support object branch changes as this never was completely memory safe.

As a special rule, the discriminator kind can also be bounded using a `case` statement. If possible values of the discriminator variable in a `case` statement branch are a subset of discriminator values for the selected object branch, the initialization is considered valid. This analysis only works for immutable discriminators of an ordinal type and disregards `elif` branches. For discriminator values with a `range` type, the compiler checks if the entire range of possible values for the discriminator value is valid for the chosen object branch.

A small example:

```
let unknownKind = nkSub

# invalid: unsafe initialization because the kind field is not statically known:
var y = Node(kind: unknownKind, strVal: "y")

var z = Node()
case unknownKind
of nkAdd, nkSub:
  # valid: possible values of this branch are a subset of nkAdd/nkSub object branch:
  z = Node(kind: unknownKind, leftOp: Node(), rightOp: Node())
else:
  echo "ignoring: ", unknownKind

# also valid, since unknownKindBounded can only contain the values nkAdd or nkSub
let unknownKindBounded = range[nkAdd..nkSub](unknownKind)
z = Node(kind: unknownKindBounded, leftOp: Node(), rightOp: Node())
```

8.18 Set type

The set type models the mathematical notion of a set. The set's basetype can only be an ordinal type of a certain size, namely:

- `int8-int16`
- `uint8/byte-uint16`
- `char`
- `enum`

or equivalent. For signed integers the set's base type is defined to be in the range `0 .. MaxSetElements-1` where `MaxSetElements` is currently always 2^16.

The reason is that sets are implemented as high performance bit vectors. Attempting to declare a set with a larger type will result in an error:

```
var s: set[int64] # Error: set is too large
```

Sets can be constructed via the set constructor: `{}` is the empty set. The empty set is type compatible with any concrete set type. The constructor can also be used to include elements (and ranges of elements):

operation	meaning
`A + B`	union of two sets
`A * B`	intersection of two sets
`A - B`	difference of two sets (A without B's elements)
`A == B`	set equality
`A <= B`	subset relation (A is subset of B or equal to B)
`A < B`	strict subset relation (A is a proper subset of B)
`e in A`	set membership (A contains element e)
`e notin A`	A does not contain element e
`contains(A, e)`	A contains element e
`card(A)`	the cardinality of A (number of elements in A)
`incl(A, elem)`	same as `A = A + {elem}`
`excl(A, elem)`	same as `A = A - {elem}`

```
type
  CharSet = set[char]
var
  x: CharSet
x = {'a'..'z', '0'..'9'} # This constructs a set that contains the
                         # letters from 'a' to 'z' and the digits
                         # from '0' to '9'
```

These operations are supported by sets:

8.18.1 Bit fields

Sets are often used to define a type for the *flags* of a procedure. This is a cleaner (and type safe) solution than defining integer constants that have to be `or`'ed together.

Enum, sets and casting can be used together as in:

```
type
  MyFlag* {.size: sizeof(cint).} = enum
    A
    B
    C
    D
  MyFlags = set[MyFlag]

proc toNum(f: MyFlags): int = cast[cint](f)
proc toFlags(v: int): MyFlags = cast[MyFlags](v)

assert toNum({}) == 0
assert toNum({A}) == 1
assert toNum({D}) == 8
assert toNum({A, C}) == 5
assert toFlags(0) == {}
assert toFlags(7) == {A, B, C}
```

Note how the set turns enum values into powers of 2.

If using enums and sets with C, use distinct cint.

For interoperability with C see also the bitsize pragma.

8.19 Reference and pointer types

References (similar to pointers in other programming languages) are a way to introduce many-to-one relationships. This means different references can point to and modify the same location in memory (also called aliasing).

Nim distinguishes between traced and untraced references. Untraced references are also called *pointers*. Traced references point to objects of a garbage collected heap, untraced references point to manually allocated objects or to objects somewhere else in memory. Thus untraced references are *unsafe*. However for certain low-level operations (accessing the hardware) untraced references are unavoidable.

Traced references are declared with the **ref** keyword, untraced references are declared with the **ptr** keyword. In general, a `ptr T` is implicitly convertible to the `pointer` type.

An empty subscript `[]` notation can be used to derefer a reference, the `addr` procedure returns the address of an item. An address is always an untraced reference. Thus the usage of `addr` is an *unsafe* feature.

The `.` (access a tuple/object field operator) and `[]` (array/string/sequence index operator) operators perform implicit dereferencing operations for reference types:

```
type
  Node = ref NodeObj
  NodeObj = object
    le, ri: Node
    data: int

var
  n: Node
new(n)
n.data = 9
# no need to write n[].data; in fact n[].data is highly discouraged!
```

Automatic dereferencing can be performed for the first argument of a routine call, but this is an experimental feature and is described here.

In order to simplify structural type checking, recursive tuples are not valid:

```
# invalid recursion
type MyTuple = tuple[a: ref MyTuple]
```

Likewise `T = ref T` is an invalid type.

As a syntactical extension `object` types can be anonymous if declared in a type section via the `ref object` or `ptr object` notations. This feature is useful if an object should only gain reference semantics:

```
type
  Node = ref object
    le, ri: Node
    data: int
```

To allocate a new traced object, the built-in procedure `new` has to be used. To deal with untraced memory, the procedures `alloc`, `dealloc` and `realloc` can be used. The documentation of the system module contains further information.

Nil —

If a reference points to *nothing*, it has the value `nil`. `nil` is the default value for all `ref` and `ptr` types. The `nil` value can also be used like any other literal value. For example, it can be used in an assignment like `myRef = nil`.

Dereferencing `nil` is an unrecoverable fatal runtime error (and not a panic).

A successful dereferencing operation `p[]` implies that `p` is not nil. This can be exploited by the implementation to optimize code like:

```
p[].field = 3
if p != nil:
  # if p were nil, ``p[]`` would have caused a crash already,
  # so we know ``p`` is always not nil here.
  action()
```

Into:

```
p[].field = 3
action()
```

Note: This is not comparable to C's "undefined behavior" for dereferencing NULL pointers.

8.20 Mixing GC'ed memory with `ptr`

Special care has to be taken if an untraced object contains traced objects like traced references, strings or sequences: in order to free everything properly, the built-in procedure `reset` has to be called before freeing the untraced memory manually:

```
type
  Data = tuple[x, y: int, s: string]

# allocate memory for Data on the heap:
var d = cast[ptr Data](alloc0(sizeof(Data)))

# create a new string on the garbage collected heap:
d.s = "abc"

# tell the GC that the string is not needed anymore:
reset(d.s)

# free the memory:
dealloc(d)
```

Without the `reset` call the memory allocated for the `d.s` string would never be freed. The example also demonstrates two important features for low level programming: the `sizeof` proc returns the size of a type or value in bytes. The `cast` operator can circumvent the type system: the compiler is forced to treat the result of the `alloc0` call (which returns an untyped pointer) as if it would have the type `ptr Data`. Casting should only be done if it is unavoidable: it breaks type safety and bugs can lead to mysterious crashes.

Note: The example only works because the memory is initialized to zero (`alloc0` instead of `alloc` does this): `d.s` is thus initialized to binary zero which the string assignment can handle. One needs to know low level details like this when mixing garbage collected data with unmanaged memory.

8.21 Procedural type

A procedural type is internally a pointer to a procedure. `nil` is an allowed value for variables of a procedural type. Nim uses procedural types to achieve functional programming techniques.

Examples:

```
proc printItem(x: int) = ...

proc forEach(c: proc (x: int) {.cdecl.}) =
  ...

forEach(printItem)  # this will NOT compile because calling conventions differ

type
  OnMouseMove = proc (x, y: int) {.closure.}

proc onMouseMove(mouseX, mouseY: int) =
  # has default calling convention
  echo "x: ", mouseX, " y: ", mouseY

proc setOnMouseMove(mouseMoveEvent: OnMouseMove) = discard

# ok, 'onMouseMove' has the default calling convention, which is compatible
# to 'closure':
setOnMouseMove(onMouseMove)
```

A subtle issue with procedural types is that the calling convention of the procedure influences the type compatibility: procedural types are only compatible if they have the same calling convention. As a special extension, a procedure of the calling convention `nimcall` can be passed to a parameter that expects a proc of the calling convention `closure`.

Nim supports these calling conventions:

nimcall is the default convention used for a Nim **proc**. It is the same as `fastcall`, but only for C compilers that support `fastcall`.

closure is the default calling convention for a **procedural type** that lacks any pragma annotations. It indicates that the procedure has a hidden implicit parameter (an *environment*). Proc vars that have the calling convention `closure` take up two machine words: One for the proc pointer and another one for the pointer to implicitly passed environment.

stdcall This is the stdcall convention as specified by Microsoft. The generated C procedure is declared with the `__stdcall` keyword.

cdecl The cdecl convention means that a procedure shall use the same convention as the C compiler. Under Windows the generated C procedure is declared with the `__cdecl` keyword.

safecall This is the safecall convention as specified by Microsoft. The generated C procedure is declared with the `__safecall` keyword. The word *safe* refers to the fact that all hardware registers shall be pushed to the hardware stack.

inline The inline convention means the the caller should not call the procedure, but inline its code directly. Note that Nim does not inline, but leaves this to the C compiler; it generates `__inline` procedures. This is only a hint for the compiler: it may completely ignore it and it may inline procedures that are not marked as `inline`.

fastcall Fastcall means different things to different C compilers. One gets whatever the C `__fastcall` means.

thiscall This is thiscall calling convention as specified by Microsoft, used on C++ class member functions on the x86 architecture

syscall The syscall convention is the same as `__syscall` in C. It is used for interrupts.

noconv The generated C code will not have any explicit calling convention and thus use the C compiler's default calling convention. This is needed because Nim's default calling convention for procedures is `fastcall` to improve speed.

Most calling conventions exist only for the Windows 32-bit platform.

The default calling convention is `nimcall`, unless it is an inner proc (a proc inside of a proc). For an inner proc an analysis is performed whether it accesses its environment. If it does so, it has the calling convention `closure`, otherwise it has the calling convention `nimcall`.

8.22 Distinct type

A `distinct` type is new type derived from a base type that is incompatible with its base type. In particular, it is an essential property of a distinct type that it **does not** imply a subtype relation between it and its base type. Explicit type conversions from a distinct type to its base type and vice versa are allowed. See also `distinctBase` to get the reverse operation.

A distinct type is an ordinal type if its base type is an ordinal type.

8.22.1 Modelling currencies

A distinct type can be used to model different physical units with a numerical base type, for example. The following example models currencies.

Different currencies should not be mixed in monetary calculations. Distinct types are a perfect tool to model different currencies:

```
type
  Dollar = distinct int
  Euro = distinct int

var
  d: Dollar
  e: Euro

echo d + 12
# Error: cannot add a number with no unit and a ``Dollar``
```

Unfortunately, `d + 12.Dollar` is not allowed either, because + is defined for `int` (among others), not for `Dollar`. So a + for dollars needs to be defined:

```
proc `+` (x, y: Dollar): Dollar =
  result = Dollar(int(x) + int(y))
```

It does not make sense to multiply a dollar with a dollar, but with a number without unit; and the same holds for division:

```
proc `*` (x: Dollar, y: int): Dollar =
  result = Dollar(int(x) * y)

proc `*` (x: int, y: Dollar): Dollar =
  result = Dollar(x * int(y))

proc `div` ...
```

This quickly gets tedious. The implementations are trivial and the compiler should not generate all this code only to optimize it away later - after all + for dollars should produce the same binary code as + for ints. The pragma borrow has been designed to solve this problem; in principle it generates the above trivial implementations:

```
proc `*` (x: Dollar, y: int): Dollar {.borrow.}
proc `*` (x: int, y: Dollar): Dollar {.borrow.}
proc `div` (x: Dollar, y: int): Dollar {.borrow.}
```

The `borrow` pragma makes the compiler use the same implementation as the proc that deals with the distinct type's base type, so no code is generated.

But it seems all this boilerplate code needs to be repeated for the `Euro` currency. This can be solved with templates20.

```
template additive(typ: typedesc) =
  proc `+` *(x, y: typ): typ {.borrow.}
  proc `-` *(x, y: typ): typ {.borrow.}

  # unary operators:
  proc `+` *(x: typ): typ {.borrow.}
  proc `-` *(x: typ): typ {.borrow.}

template multiplicative(typ, base: typedesc) =
  proc `*` *(x: typ, y: base): typ {.borrow.}
  proc `*` *(x: base, y: typ): typ {.borrow.}
  proc `div` *(x: typ, y: base): typ {.borrow.}
  proc `mod` *(x: typ, y: base): typ {.borrow.}

template comparable(typ: typedesc) =
  proc `<` * (x, y: typ): bool {.borrow.}
  proc `<=` * (x, y: typ): bool {.borrow.}
  proc `==` * (x, y: typ): bool {.borrow.}

template defineCurrency(typ, base: untyped) =
  type
    typ* = distinct base
  additive(typ)
  multiplicative(typ, base)
  comparable(typ)

defineCurrency(Dollar, int)
defineCurrency(Euro, int)
```

The borrow pragma can also be used to annotate the distinct type to allow certain builtin operations to be lifted:

```
type
  Foo = object
    a, b: int
    s: string
```

```
  Bar {.borrow: `.`.} = distinct Foo

var bb: ref Bar
new bb
# field access now valid
bb.a = 90
bb.s = "abc"
```

Currently only the dot accessor can be borrowed in this way.

8.22.2 Avoiding SQL injection attacks

An SQL statement that is passed from Nim to an SQL database might be modelled as a string. However, using string templates and filling in the values is vulnerable to the famous SQL injection attack:

```
import strutils

proc query(db: DbHandle, statement: string) = ...

var
  username: string

db.query("SELECT FROM users WHERE name = '$1'" % username)
# Horrible security hole, but the compiler does not mind!
```

This can be avoided by distinguishing strings that contain SQL from strings that don't. Distinct types provide a means to introduce a new string type `SQL` that is incompatible with `string`:

```
type
  SQL = distinct string

proc query(db: DbHandle, statement: SQL) = ...

var
  username: string

db.query("SELECT FROM users WHERE name = '$1'" % username)
# Static error: `query` expects an SQL string!
```

It is an essential property of abstract types that they **do not** imply a subtype relation between the abstract type and its base type. Explicit type conversions from `string` to `SQL` are allowed:

```
import strutils, sequtils

proc properQuote(s: string): SQL =
  # quotes a string properly for an SQL statement
  return SQL(s)

proc `%` (frmt: SQL, values: openarray[string]): SQL =
  # quote each argument:
  let v = values.mapIt(SQL, properQuote(it))
  # we need a temporary type for the type conversion :-(
  type StrSeq = seq[string]
  # call strutils.`%`:
  result = SQL(string(frmt) % StrSeq(v))

db.query("SELECT FROM users WHERE name = '$1'".SQL % [username])
```

Now we have compile-time checking against SQL injection attacks. Since `"".SQL` is transformed to `SQL("")` no new syntax is needed for nice looking `SQL` string literals. The hypothetical `SQL` type actually exists in the library as the SqlQuery type of modules like db_sqlite.

8.23 Auto type

The `auto` type can only be used for return types and parameters. For return types it causes the compiler to infer the type from the routine body:

```
proc returnsInt(): auto = 1984
```

For parameters it currently creates implicitly generic routines:

```
proc foo(a, b: auto) = discard
```

Is the same as:

```
proc foo[T1, T2](a: T1, b: T2) = discard
```

However later versions of the language might change this to mean "infer the parameters' types from the body". Then the above `foo` would be rejected as the parameters' types can not be inferred from an empty `discard` statement.

9 Type relations

The following section defines several relations on types that are needed to describe the type checking done by the compiler.

9.1 Type equality

Nim uses structural type equivalence for most types. Only for objects, enumerations and distinct types name equivalence is used. The following algorithm, *in pseudo-code*, determines type equality:

```
proc typeEqualsAux(a, b: PType,
                   s: var HashSet[(PType, PType)]): bool =
  if (a,b) in s: return true
  incl(s, (a,b))
  if a.kind == b.kind:
    case a.kind
    of int, intXX, float, floatXX, char, string, cstring, pointer,
        bool, nil, void:
      # leaf type: kinds identical; nothing more to check
      result = true
    of ref, ptr, var, set, seq, openarray:
      result = typeEqualsAux(a.baseType, b.baseType, s)
    of range:
      result = typeEqualsAux(a.baseType, b.baseType, s) and
        (a.rangeA == b.rangeA) and (a.rangeB == b.rangeB)
    of array:
      result = typeEqualsAux(a.baseType, b.baseType, s) and
               typeEqualsAux(a.indexType, b.indexType, s)
    of tuple:
      if a.tupleLen == b.tupleLen:
        for i in 0..a.tupleLen-1:
          if not typeEqualsAux(a[i], b[i], s): return false
        result = true
    of object, enum, distinct:
      result = a == b
    of proc:
      result = typeEqualsAux(a.parameterTuple, b.parameterTuple, s) and
               typeEqualsAux(a.resultType, b.resultType, s) and
               a.callingConvention == b.callingConvention

proc typeEquals(a, b: PType): bool =
  var s: HashSet[(PType, PType)] = {}
  result = typeEqualsAux(a, b, s)
```

Since types are graphs which can have cycles, the above algorithm needs an auxiliary set `s` to detect this case.

9.2 Type equality modulo type distinction

The following algorithm (in pseudo-code) determines whether two types are equal with no respect to `distinct` types. For brevity the cycle check with an auxiliary set `s` is omitted:

```
proc typeEqualsOrDistinct(a, b: PType): bool =
  if a.kind == b.kind:
    case a.kind
    of int, intXX, float, floatXX, char, string, cstring, pointer,
        bool, nil, void:
      # leaf type: kinds identical; nothing more to check
      result = true
    of ref, ptr, var, set, seq, openarray:
      result = typeEqualsOrDistinct(a.baseType, b.baseType)
    of range:
      result = typeEqualsOrDistinct(a.baseType, b.baseType) and
        (a.rangeA == b.rangeA) and (a.rangeB == b.rangeB)
    of array:
      result = typeEqualsOrDistinct(a.baseType, b.baseType) and
               typeEqualsOrDistinct(a.indexType, b.indexType)
    of tuple:
      if a.tupleLen == b.tupleLen:
        for i in 0..a.tupleLen-1:
          if not typeEqualsOrDistinct(a[i], b[i]): return false
        result = true
    of distinct:
      result = typeEqualsOrDistinct(a.baseType, b.baseType)
    of object, enum:
      result = a == b
    of proc:
      result = typeEqualsOrDistinct(a.parameterTuple, b.parameterTuple) and
               typeEqualsOrDistinct(a.resultType, b.resultType) and
               a.callingConvention == b.callingConvention
  elif a.kind == distinct:
    result = typeEqualsOrDistinct(a.baseType, b)
  elif b.kind == distinct:
    result = typeEqualsOrDistinct(a, b.baseType)
```

9.3 Subtype relation

If object `a` inherits from `b`, `a` is a subtype of `b`. This subtype relation is extended to the types `var`, `ref`, `ptr`:

```
proc isSubtype(a, b: PType): bool =
  if a.kind == b.kind:
    case a.kind
    of object:
      var aa = a.baseType
      while aa != nil and aa != b: aa = aa.baseType
      result = aa == b
    of var, ref, ptr:
      result = isSubtype(a.baseType, b.baseType)
```

9.4 Convertible relation

A type `a` is **implicitly** convertible to type `b` iff the following algorithm returns true:

```
proc isImplicitlyConvertible(a, b: PType): bool =
  if isSubtype(a, b) or isCovariant(a, b):
    return true
  if isIntLiteral(a):
    return b in {int8, int16, int32, int64, int, uint, uint8, uint16,
                 uint32, uint64, float32, float64}
  case a.kind
  of int:     result = b in {int32, int64}
  of int8:    result = b in {int16, int32, int64, int}
  of int16:   result = b in {int32, int64, int}
  of int32:   result = b in {int64, int}
```

```
  of uint:    result = b in {uint32, uint64}
  of uint8:   result = b in {uint16, uint32, uint64}
  of uint16:  result = b in {uint32, uint64}
  of uint32:  result = b in {uint64}
  of float32: result = b in {float64}
  of float64: result = b in {float32}
  of seq:
    result = b == openArray and typeEquals(a.baseType, b.baseType)
  of array:
    result = b == openArray and typeEquals(a.baseType, b.baseType)
    if a.baseType == char and a.indexType.rangeA == 0:
      result = b == cstring
  of cstring, ptr:
    result = b == pointer
  of string:
    result = b == cstring
```

Implicit conversions are also performed for Nim's `range` type constructor.

Let `a0`, `b0` of type `T`.

Let `A = range[a0..b0]` be the argument's type, `F` the formal parameter's type. Then an implicit conversion from `A` to `F` exists if `a0 >= low(F) and b0 <= high(F)` and both `T` and `F` are signed integers or if both are unsigned integers.

A type `a` is **explicitly** convertible to type `b` iff the following algorithm returns true:

```
proc isIntegralType(t: PType): bool =
  result = isOrdinal(t) or t.kind in {float, float32, float64}

proc isExplicitlyConvertible(a, b: PType): bool =
  result = false
  if isImplicitlyConvertible(a, b): return true
  if typeEqualsOrDistinct(a, b): return true
  if isIntegralType(a) and isIntegralType(b): return true
  if isSubtype(a, b) or isSubtype(b, a): return true
```

The convertible relation can be relaxed by a user-defined type converter.

```
converter toInt(x: char): int = result = ord(x)

var
  x: int
  chr: char = 'a'

# implicit conversion magic happens here
x = chr
echo x # => 97
# one can use the explicit form too
x = chr.toInt
echo x # => 97
```

The type conversion `T(a)` is an L-value if `a` is an L-value and `typeEqualsOrDistinct(T, typeof(a))` holds.

9.5 Assignment compatibility

An expression `b` can be assigned to an expression `a` iff `a` is an `l-value` and `isImplicitlyConvertible(b.typ, a.typ)` holds.

10 Overloading resolution

In a call `p(args)` the routine `p` that matches best is selected. If multiple routines match equally well, the ambiguity is reported during semantic analysis.

Every arg in args needs to match. There are multiple different categories how an argument can match. Let `f` be the formal parameter's type and `a` the type of the argument.

1. Exact match: `a` and `f` are of the same type.

2. Literal match: `a` is an integer literal of value `v` and `f` is a signed or unsigned integer type and `v` is in `f`'s range. Or: `a` is a floating point literal of value `v` and `f` is a floating point type and `v` is in `f`'s range.

3. Generic match: `f` is a generic type and `a` matches, for instance `a` is `int` and `f` is a generic (constrained) parameter type (like in `[T]` or `[T: int|char]`.

4. Subrange or subtype match: `a` is a `range[T]` and `T` matches `f` exactly. Or: `a` is a subtype of `f`.

5. Integral conversion match: `a` is convertible to `f` and `f` and `a` is some integer or floating point type.

6. Conversion match: `a` is convertible to `f`, possibly via a user defined `converter`.

These matching categories have a priority: An exact match is better than a literal match and that is better than a generic match etc. In the following `count(p, m)` counts the number of matches of the matching category `m` for the routine `p`.

A routine `p` matches better than a routine `q` if the following algorithm returns true:

```
for each matching category m in ["exact match", "literal match",
                                "generic match", "subtype match",
                                "integral match", "conversion match"]:
  if count(p, m) > count(q, m): return true
  elif count(p, m) == count(q, m):
    discard "continue with next category m"
  else:
    return false
return "ambiguous"
```

Some examples:

```
proc takesInt(x: int) = echo "int"
proc takesInt[T](x: T) = echo "T"
proc takesInt(x: int16) = echo "int16"

takesInt(4) # "int"
var x: int32
takesInt(x) # "T"
var y: int16
takesInt(y) # "int16"
var z: range[0..4] = 0
takesInt(z) # "T"
```

If this algorithm returns "ambiguous" further disambiguation is performed: If the argument `a` matches both the parameter type `f` of `p` and `g` of `q` via a subtyping relation, the inheritance depth is taken into account:

```
type
  A = object of RootObj
  B = object of A
  C = object of B

proc p(obj: A) =
  echo "A"

proc p(obj: B) =
  echo "B"

var c = C()
# not ambiguous, calls 'B', not 'A' since B is a subtype of A
# but not vice versa:
p(c)

proc pp(obj: A, obj2: B) = echo "A B"
proc pp(obj: B, obj2: A) = echo "B A"

# but this is ambiguous:
pp(c, c)
```

Likewise for generic matches the most specialized generic type (that still matches) is preferred:

```
proc gen[T](x: ref ref T) = echo "ref ref T"
proc gen[T](x: ref T) = echo "ref T"
proc gen[T](x: T) = echo "T"

var ri: ref int
gen(ri) # "ref T"
```

10.1 Overloading based on 'var T' / 'out T'

If the formal parameter `f` is of type `var T` (or `out T`) in addition to the ordinary type checking, the argument is checked to be an l-value. `var T` (or `out T`) matches better than just `T` then.

```
proc sayHi(x: int): string =
  # matches a non-var int
  result = $x
proc sayHi(x: var int): string =
  # matches a var int
  result = $(x + 10)

proc sayHello(x: int) =
  var m = x # a mutable version of x
  echo sayHi(x) # matches the non-var version of sayHi
  echo sayHi(m) # matches the var version of sayHi

sayHello(3) # 3
            # 13
```

An l-value matches `var T` and `out T` equally well, hence the following is ambiguous:

```
proc p(x: out string) = x = ""
proc p(x: var string) = x = ""
var v: string
p(v) # ambiguous
```

10.2 Lazy type resolution for untyped

Note: An unresolved expression is an expression for which no symbol lookups and no type checking have been performed.

Since templates and macros that are not declared as `immediate` participate in overloading resolution it's essential to have a way to pass unresolved expressions to a template or macro. This is what the meta-type `untyped` accomplishes:

```
template rem(x: untyped) = discard

rem unresolvedExpression(undeclaredIdentifier)
```

A parameter of type `untyped` always matches any argument (as long as there is any argument passed to it).

But one has to watch out because other overloads might trigger the argument's resolution:

```
template rem(x: untyped) = discard
proc rem[T](x: T) = discard

# undeclared identifier: 'unresolvedExpression'
rem unresolvedExpression(undeclaredIdentifier)
```

`untyped` and `varargs[untyped]` are the only metatype that are lazy in this sense, the other metatypes `typed` and `typedesc` are not lazy.

10.3 Varargs matching

See Varargs.

11 Statements and expressions

Nim uses the common statement/expression paradigm: Statements do not produce a value in contrast to expressions. However, some expressions are statements.

Statements are separated into simple statements and complex statements. Simple statements are statements that cannot contain other statements like assignments, calls or the `return` statement; complex statements can contain other statements. To avoid the dangling else problem, complex statements always have to be indented. The details can be found in the grammar.

11.1 Statement list expression

Statements can also occur in an expression context that looks like `(stmt1; stmt2; ...; ex)`. This is called an statement list expression or `(;)`. The type of `(stmt1; stmt2; ...; ex)` is the type of `ex`. All the other statements must be of type `void`. (One can use `discard` to produce a `void` type.) `(;)` does not introduce a new scope.

11.2 Discard statement

Example:

```
proc p(x, y: int): int =
  result = x + y

discard p(3, 4) # discard the return value of `p`
```

The `discard` statement evaluates its expression for side-effects and throws the expression's resulting value away, and should only be used when ignoring this value is known not to cause problems.

Ignoring the return value of a procedure without using a discard statement is a static error.

The return value can be ignored implicitly if the called proc/iterator has been declared with the discardable pragma:

```
proc p(x, y: int): int {.discardable.} =
  result = x + y

p(3, 4) # now valid
```

An empty `discard` statement is often used as a null statement:

```
proc classify(s: string) =
  case s[0]
  of SymChars, '_': echo "an identifier"
  of '0'..'9': echo "a number"
  else: discard
```

11.3 Void context

In a list of statements every expression except the last one needs to have the type `void`. In addition to this rule an assignment to the builtin `result` symbol also triggers a mandatory `void` context for the subsequent expressions:

```
proc invalid*(): string =
  result = "foo"
  "invalid"  # Error: value of type 'string' has to be discarded

proc valid*(): string =
  let x = 317
  "valid"
```

Type	default value
any integer type	0
any float	0.0
char	'\0'
bool	false
ref or pointer type	nil
procedural type	nil
sequence	`@[]`
string	`""`
tuple[x: A, y: B, ...]	(default(A), default(B), ...) (analogous for objects)
array[0..., T]	[default(T), ...]
range[T]	default(T); this may be out of the valid range
T = enum	cast[T](0); this may be an invalid value

11.4 Var statement

Var statements declare new local and global variables and initialize them. A comma separated list of variables can be used to specify variables of the same type:

```
var
  a: int = 0
  x, y, z: int
```

If an initializer is given the type can be omitted: the variable is then of the same type as the initializing expression. Variables are always initialized with a default value if there is no initializing expression. The default value depends on the type and is always a zero in binary.

The implicit initialization can be avoided for optimization reasons with the noinit pragma:

```
var
  a {.noInit.}: array[0..1023, char]
```

If a proc is annotated with the `noinit` pragma this refers to its implicit `result` variable:

```
proc returnUndefinedValue: int {.noinit.} = discard
```

The implicit initialization can be also prevented by the requiresInit type pragma. The compiler requires an explicit initialization for the object and all of its fields. However it does a control flow analysis to prove the variable has been initialized and does not rely on syntactic properties:

```
type
  MyObject = object {.requiresInit.}

proc p() =
  # the following is valid:
  var x: MyObject
  if someCondition():
    x = a()
  else:
    x = a()
  # use x
```

11.5 Let statement

A `let` statement declares new local and global single assignment variables and binds a value to them. The syntax is the same as that of the `var` statement, except that the keyword `var` is replaced by the keyword `let`. Let variables are not l-values and can thus not be passed to `var` parameters nor can their address be taken. They cannot be assigned new values.

For let variables the same pragmas are available as for ordinary variables.

As `let` statements are immutable after creation they need to define a value when they are declared. The only exception to this is if the `{.importc.}` pragma (or any of the other `importX` pragmas) is applied, in this case the value is expected to come from native code, typically a C/C++ `const`.

11.6 Tuple unpacking

In a `var` or `let` statement tuple unpacking can be performed. The special identifier _ can be used to ignore some parts of the tuple:

```
proc returnsTuple(): (int, int, int) = (4, 2, 3)

let (x, _, z) = returnsTuple()
```

11.7 Const section

A const section declares constants whose values are constant expressions:

```
import strutils
const
  roundPi = 3.1415
  constEval = contains("abc", 'b') # computed at compile time!
```

Once declared, a constant's symbol can be used as a constant expression.

See Constants and Constant Expressions for details.

11.8 Static statement/expression

A static statement/expression explicitly requires compile-time execution. Even some code that has side effects is permitted in a static block:

```
static:
  echo "echo at compile time"
```

There are limitations on what Nim code can be executed at compile time; see Restrictions on Compile-Time Execution for details. It's a static error if the compiler cannot execute the block at compile time.

11.9 If statement

Example:

```
var name = readLine(stdin)

if name == "Andreas":
  echo "What a nice name!"
elif name == "":
  echo "Don't you have a name?"
else:
  echo "Boring name..."
```

The `if` statement is a simple way to make a branch in the control flow: The expression after the keyword `if` is evaluated, if it is true the corresponding statements after the `:` are executed. Otherwise the expression after the `elif` is evaluated (if there is an `elif` branch), if it is true the corresponding statements after the `:` are executed. This goes on until the last `elif`. If all conditions fail, the `else` part is executed. If there is no `else` part, execution continues with the next statement.

In `if` statements new scopes begin immediately after the `if`/`elif`/`else` keywords and ends after the corresponding *then* block. For visualization purposes the scopes have been enclosed in `{| |}` in the following example:

```
if {| (let m = input =~ re"(\w+)=\w+"; m.isMatch):
  echo "key ", m[0], " value ", m[1]  |}
elif {| (let m = input =~ re""; m.isMatch):
  echo "new m in this scope"  |}
else: {|
  echo "m not declared here"  |}
```

11.10 Case statement

Example:

```
case readline(stdin)
of "delete-everything", "restart-computer":
  echo "permission denied"
of "go-for-a-walk":     echo "please yourself"
else:                   echo "unknown command"

# indentation of the branches is also allowed; and so is an optional colon
# after the selecting expression:
case readline(stdin):
  of "delete-everything", "restart-computer":
    echo "permission denied"
  of "go-for-a-walk":     echo "please yourself"
  else:                   echo "unknown command"
```

The `case` statement is similar to the if statement, but it represents a multi-branch selection. The expression after the keyword `case` is evaluated and if its value is in a *slicelist* the corresponding statements (after the `of` keyword) are executed. If the value is not in any given *slicelist* the `else` part is executed. If there is no `else` part and not all possible values that `expr` can hold occur in a `slicelist`, a static error occurs. This holds only for expressions of ordinal types. "All possible values" of `expr` are determined by `expr`'s type. To suppress the static error an `else` part with an empty `discard` statement should be used.

For non ordinal types it is not possible to list every possible value and so these always require an `else` part.

Because case statements are checked for exhaustiveness during semantic analysis, the value in every `of` branch must be a constant expression. This restriction also allows the compiler to generate more performant code.

As a special semantic extension, an expression in an `of` branch of a case statement may evaluate to a set or array constructor; the set or array is then expanded into a list of its elements:

```
const
  SymChars: set[char] = {'a'..'z', 'A'..'Z', '\x80'..'\xFF'}

proc classify(s: string) =
  case s[0]
  of SymChars, '_': echo "an identifier"
  of '0'..'9': echo "a number"
  else: echo "other"

# is equivalent to:
proc classify(s: string) =
  case s[0]
  of 'a'..'z', 'A'..'Z', '\x80'..'\xFF', '_': echo "an identifier"
  of '0'..'9': echo "a number"
  else: echo "other"
```

The `case` statement doesn't produce an l-value, so the following example won't work:

```
type
  Foo = ref object
    x: seq[string]

proc get_x(x: Foo): var seq[string] =
  # doesn't work
  case true
  of true:
    x.x
  else:
    x.x

var foo = Foo(x: @[])
foo.get_x().add("asd")
```

This can be fixed by explicitly using `return`:

```
proc get_x(x: Foo): var seq[string] =
  case true
  of true:
    return x.x
  else:
    return x.x
```

11.11 When statement

Example:

```
when sizeof(int) == 2:
  echo "running on a 16 bit system!"
elif sizeof(int) == 4:
  echo "running on a 32 bit system!"
elif sizeof(int) == 8:
  echo "running on a 64 bit system!"
else:
  echo "cannot happen!"
```

The `when` statement is almost identical to the `if` statement with some exceptions:

- Each condition (`expr`) has to be a constant expression (of type `bool`).
- The statements do not open a new scope.
- The statements that belong to the expression that evaluated to true are translated by the compiler, the other statements are not checked for semantics! However, each condition is checked for semantics.

The `when` statement enables conditional compilation techniques. As a special syntactic extension, the `when` construct is also available within `object` definitions.

11.12 When nimvm statement

`nimvm` is a special symbol, that may be used as expression of `when nimvm` statement to differentiate execution path between compile time and the executable.

Example:

```
proc someProcThatMayRunInCompileTime(): bool =
  when nimvm:
    # This branch is taken at compile time.
    result = true
  else:
    # This branch is taken in the executable.
    result = false
const ctValue = someProcThatMayRunInCompileTime()
let rtValue = someProcThatMayRunInCompileTime()
assert(ctValue == true)
assert(rtValue == false)
```

`when nimvm` statement must meet the following requirements:

- Its expression must always be `nimvm`. More complex expressions are not allowed.
- It must not contain `elif` branches.
- It must contain `else` branch.
- Code in branches must not affect semantics of the code that follows the `when nimvm` statement. E.g. it must not define symbols that are used in the following code.

11.13 Return statement

Example:

```
return 40+2
```

The `return` statement ends the execution of the current procedure. It is only allowed in procedures. If there is an `expr`, this is syntactic sugar for:

```
result = expr
return result
```

`return` without an expression is a short notation for `return result` if the proc has a return type. The result variable is always the return value of the procedure. It is automatically declared by the compiler. As all variables, `result` is initialized to (binary) zero:

```
proc returnZero(): int =
  # implicitly returns 0
```

11.14 Yield statement

Example:

```
yield (1, 2, 3)
```

The `yield` statement is used instead of the `return` statement in iterators. It is only valid in iterators. Execution is returned to the body of the for loop that called the iterator. Yield does not end the iteration process, but execution is passed back to the iterator if the next iteration starts. See the section about iterators (Iterators and the for statement14) for further information.

11.15 Block statement

Example:

```
var found = false
block myblock:
  for i in 0..3:
    for j in 0..3:
      if a[j][i] == 7:
        found = true
        break myblock # leave the block, in this case both for-loops
echo found
```

The block statement is a means to group statements to a (named) `block`. Inside the block, the `break` statement is allowed to leave the block immediately. A `break` statement can contain a name of a surrounding block to specify which block is to leave.

11.16 Break statement

Example:

```
break
```

The `break` statement is used to leave a block immediately. If `symbol` is given, it is the name of the enclosing block that is to leave. If it is absent, the innermost block is left.

11.17 While statement

Example:

```
echo "Please tell me your password:"
var pw = readLine(stdin)
while pw != "12345":
  echo "Wrong password! Next try:"
  pw = readLine(stdin)
```

The `while` statement is executed until the `expr` evaluates to false. Endless loops are no error. `while` statements open an `implicit block`, so that they can be left with a `break` statement.

11.18 Continue statement

A `continue` statement leads to the immediate next iteration of the surrounding loop construct. It is only allowed within a loop. A continue statement is syntactic sugar for a nested block:

```
while expr1:
  stmt1
  continue
  stmt2
```

Is equivalent to:

```
while expr1:
  block myBlockName:
    stmt1
    break myBlockName
    stmt2
```

11.19 Assembler statement

The direct embedding of assembler code into Nim code is supported by the unsafe `asm` statement. Identifiers in the assembler code that refer to Nim identifiers shall be enclosed in a special character which can be specified in the statement's pragmas. The default special character is `'`'`:

```
{.push stackTrace:off.}
proc addInt(a, b: int): int =
  # a in eax, and b in edx
  asm """      mov eax, `a`      add eax, `b`      jno theEnd      call `raiseOverflow`    theEnd:  """
{.pop.}
```

If the GNU assembler is used, quotes and newlines are inserted automatically:

```
proc addInt(a, b: int): int =
  asm """    addl %%ecx, %%eax    jno 1    call `raiseOverflow`    1:    :"=a"(`result`)    :"a"(`a`), "c"(`b`)
```

Instead of:

```
proc addInt(a, b: int): int =
  asm """    "addl %%ecx, %%eax\n"    "jno 1\n"    "call `raiseOverflow`\n"    "1: \n"    :"=a"(`result`)    :"a"
```

11.20 Using statement

The using statement provides syntactic convenience in modules where the same parameter names and types are used over and over. Instead of:

```
proc foo(c: Context; n: Node) = ...
proc bar(c: Context; n: Node, counter: int) = ...
proc baz(c: Context; n: Node) = ...
```

One can tell the compiler about the convention that a parameter of name `c` should default to type `Context`, `n` should default to `Node` etc.:

```
using
  c: Context
  n: Node
  counter: int

proc foo(c, n) = ...
proc bar(c, n, counter) = ...
proc baz(c, n) = ...

proc mixedMode(c, n; x, y: int) =
  # 'c' is inferred to be of the type 'Context'
  # 'n' is inferred to be of the type 'Node'
  # But 'x' and 'y' are of type 'int'.
```

The `using` section uses the same indentation based grouping syntax as a `var` or `let` section.

Note that `using` is not applied for `template` since untyped template parameters default to the type `system.untyped`.

Mixing parameters that should use the `using` declaration with parameters that are explicitly typed is possible and requires a semicolon between them.

11.21 If expression

An `if expression` is almost like an if statement, but it is an expression. This feature is similar to `ternary operators` in other languages. Example:

```
var y = if x > 8: 9 else: 10
```

An if expression always results in a value, so the `else` part is required. `Elif` parts are also allowed.

11.22 When expression

Just like an `if expression`, but corresponding to the when statement.

11.23 Case expression

The `case expression` is again very similar to the case statement:

```
var favoriteFood = case animal
  of "dog": "bones"
  of "cat": "mice"
  elif animal.endsWith"whale": "plankton"
  else:
    echo "I'm not sure what to serve, but everybody loves ice cream"
    "ice cream"
```

As seen in the above example, the case expression can also introduce side effects. When multiple statements are given for a branch, Nim will use the last expression as the result value.

11.24 Block expression

A `block expression` is almost like a block statement, but it is an expression that uses last expression under the block as the value. It is similar to the statement list expression, but the statement list expression does not open new block scope.

```
let a = block:
  var fib = @[0, 1]
  for i in 0..10:
    fib.add fib[^1] + fib[^2]
  fib
```

11.25 Table constructor

A table constructor is syntactic sugar for an array constructor:

```
{"key1": "value1", "key2", "key3": "value2"}

# is the same as:
[("key1", "value1"), ("key2", "value2"), ("key3", "value2")]
```

The empty table can be written `{:}` (in contrast to the empty set which is `{}`) which is thus another way to write as the empty array constructor `[]`. This slightly unusual way of supporting tables has lots of advantages:

- The order of the (key,value)-pairs is preserved, thus it is easy to support ordered dicts with for example `{key: val}.newOrderedTable`.
- A table literal can be put into a `const` section and the compiler can easily put it into the executable's data section just like it can for arrays and the generated data section requires a minimal amount of memory.
- Every table implementation is treated equal syntactically.
- Apart from the minimal syntactic sugar the language core does not need to know about tables.

11.26 Type conversions

Syntactically a *type conversion* is like a procedure call, but a type name replaces the procedure name. A type conversion is always safe in the sense that a failure to convert a type to another results in an exception (if it cannot be determined statically).

Ordinary procs are often preferred over type conversions in Nim: For instance, `$` is the `toString` operator by convention and `toFloat` and `toInt` can be used to convert from floating point to integer or vice versa.

A type conversion can also be used to disambiguate overloaded routines:

```
proc p(x: int) = echo "int"
proc p(x: string) = echo "string"

let procVar = (proc(x: string))(p)
procVar("a")
```

Since operations on unsigned numbers wrap around and are unchecked so are type conversion to unsigned integers and between unsigned integers. The rationale for this is mostly better interoperability with the C Programming language when algorithms are ported from C to Nim.

Exception: Values that are converted to an unsigned type at compile time are checked so that code like `byte(-1)` does not compile.

Note: Historically the operations were unchecked and the conversions were sometimes checked but starting with the revision 1.0.4 of this document and the language implementation the conversions too are now *always unchecked*.

11.27 Type casts

Type casts are a crude mechanism to interpret the bit pattern of an expression as if it would be of another type. Type casts are only needed for low-level programming and are inherently unsafe.

```
cast[int](x)
```

The target type of a cast must be a concrete type, for instance, a target type that is a type class (which is non-concrete) would be invalid:

```
type Foo = int or float
var x = cast[Foo](1) # Error: cannot cast to a non concrete type: 'Foo'
```

Type casts should not be confused with *type conversions,* as mentioned in the prior section. Unlike type conversions, a type cast cannot change the underlying bit pattern of the data being casted (aside from that the size of the target type may differ from the source type). Casting resembles *type punning* in other languages or C++'s `reinterpret_cast` and `bit_cast` features.

11.28 The addr operator

The `addr` operator returns the address of an l-value. If the type of the location is `T`, the `addr` operator result is of the type `ptr T`. An address is always an untraced reference. Taking the address of an object that resides on the stack is **unsafe**, as the pointer may live longer than the object on the stack and can thus reference a non-existing object. One can get the address of variables, but one can't use it on variables declared through `let` statements:

```
let t1 = "Hello"
var
  t2 = t1
  t3 : pointer = addr(t2)
echo repr(addr(t2))
# --> ref 0x7fff6b71b670 --> 0x10bb81050"Hello"
echo cast[ptr string](t3)[]
# --> Hello
# The following line doesn't compile:
echo repr(addr(t1))
# Error: expression has no address
```

11.29 The unsafeAddr operator

For easier interoperability with other compiled languages such as C, retrieving the address of a `let` variable, a parameter or a `for` loop variable, the `unsafeAddr` operation can be used:

```
let myArray = [1, 2, 3]
foreignProcThatTakesAnAddr(unsafeAddr myArray)
```

12 Procedures

What most programming languages call methods or functions are called procedures in Nim. A procedure declaration consists of an identifier, zero or more formal parameters, a return value type and a block of code. Formal parameters are declared as a list of identifiers separated by either comma or semicolon. A parameter is given a type by `: typename`. The type applies to all parameters immediately before it, until either the beginning of the parameter list, a semicolon separator or an already typed parameter, is reached. The semicolon can be used to make separation of types and subsequent identifiers more distinct.

```
# Using only commas
proc foo(a, b: int, c, d: bool): int

# Using semicolon for visual distinction
proc foo(a, b: int; c, d: bool): int

# Will fail: a is untyped since ';' stops type propagation.
proc foo(a; b: int; c, d: bool): int
```

A parameter may be declared with a default value which is used if the caller does not provide a value for the argument.

```
# b is optional with 47 as its default value
proc foo(a: int, b: int = 47): int
```

Parameters can be declared mutable and so allow the proc to modify those arguments, by using the type modifier `var`.

```
# "returning" a value to the caller through the 2nd argument
# Notice that the function uses no actual return value at all (ie void)
proc foo(inp: int, outp: var int) =
  outp = inp + 47
```

If the proc declaration has no body, it is a forward declaration. If the proc returns a value, the procedure body can access an implicitly declared variable named result that represents the return value. Procs can be overloaded. The overloading resolution algorithm determines which proc is the best match for the arguments. Example:

```
proc toLower(c: char): char = # toLower for characters
  if c in {'A'..'Z'}:
    result = chr(ord(c) + (ord('a') - ord('A')))
  else:
    result = c

proc toLower(s: string): string = # toLower for strings
  result = newString(len(s))
  for i in 0..len(s) - 1:
    result[i] = toLower(s[i]) # calls toLower for characters; no recursion!
```

Calling a procedure can be done in many different ways:

```
proc callme(x, y: int, s: string = "", c: char, b: bool = false) = ...

# call with positional arguments      # parameter bindings:
callme(0, 1, "abc", '\t', true)       # (x=0, y=1, s="abc", c='\t', b=true)
# call with named and positional arguments:
callme(y=1, x=0, "abd", '\t')         # (x=0, y=1, s="abd", c='\t', b=false)
# call with named arguments (order is not relevant):
callme(c='\t', y=1, x=0)              # (x=0, y=1, s="", c='\t', b=false)
# call as a command statement: no () needed:
callme 0, 1, "abc", '\t'              # (x=0, y=1, s="abc", c='\t', b=false)
```

A procedure may call itself recursively.

Operators are procedures with a special operator symbol as identifier:

```
proc `$` (x: int): string =
  # converts an integer to a string; this is a prefix operator.
  result = intToStr(x)
```

Operators with one parameter are prefix operators, operators with two parameters are infix operators. (However, the parser distinguishes these from the operator's position within an expression.) There is no way to declare postfix operators: all postfix operators are built-in and handled by the grammar explicitly.

Any operator can be called like an ordinary proc with the 'opr' notation. (Thus an operator can have more than two parameters):

```
proc `*+` (a, b, c: int): int =
  # Multiply and add
  result = a * b + c

assert `*+`(3, 4, 6) == `+`(`*`(a, b), c)
```

12.1 Export marker

If a declared symbol is marked with an asterisk it is exported from the current module:

```
proc exportedEcho*(s: string) = echo s
proc `*`*(a: string; b: int): string =
  result = newStringOfCap(a.len * b)
  for i in 1..b: result.add a

var exportedVar*: int
const exportedConst* = 78
type
  ExportedType* = object
    exportedField*: int
```

12.2 Method call syntax

For object oriented programming, the syntax `obj.method(args)` can be used instead of `method(obj, args)`. The parentheses can be omitted if there are no remaining arguments: `obj.len` (instead of `len(obj)`).

This method call syntax is not restricted to objects, it can be used to supply any type of first argument for procedures:

```
echo "abc".len # is the same as echo len "abc"
echo "abc".toUpper()
echo {'a', 'b', 'c'}.card
stdout.writeLine("Hallo") # the same as writeLine(stdout, "Hallo")
```

Another way to look at the method call syntax is that it provides the missing postfix notation.

The method call syntax conflicts with explicit generic instantiations: `p[T](x)` cannot be written as `x.p[T]` because `x.p[T]` is always parsed as `(x.p)[T]`.

See also: Limitations of the method call syntax.

The `[: ]` notation has been designed to mitigate this issue: `x.p[:T]` is rewritten by the parser to `p[T](x)`, `x.p[:T](y)` is rewritten to `p[T](x, y)`. Note that `[: ]` has no AST representation, the rewrite is performed directly in the parsing step.

12.3 Properties

Nim has no need for *get-properties*: Ordinary get-procedures that are called with the *method call syntax* achieve the same. But setting a value is different; for this a special setter syntax is needed:

```
# Module asocket
type
  Socket* = ref object of RootObj
    host: int # cannot be accessed from the outside of the module

proc `host=`*(s: var Socket, value: int) {.inline.} =
  ## setter of hostAddr.
  ## This accesses the 'host' field and is not a recursive call to
  ## ``host=`` because the builtin dot access is preferred if it is
  ## available:
  s.host = value

proc host*(s: Socket): int {.inline.} =
  ## getter of hostAddr
  ## This accesses the 'host' field and is not a recursive call to
  ## ``host`` because the builtin dot access is preferred if it is
  ## available:
  s.host
```

```
# module B
import asocket
var s: Socket
new s
s.host = 34  # same as `host=`(s, 34)
```

A proc defined as `f=` (with the trailing `=`) is called a setter. A setter can be called explicitly via the common backticks notation:

```
proc `f=`(x: MyObject; value: string) =
  discard

`f=`(myObject, "value")
```

`f=` can be called implicitly in the pattern `x.f = value` if and only if the type of `x` does not have a field named `f` or if `f` is not visible in the current module. These rules ensure that object fields and accessors can have the same name. Within the module `x.f` is then always interpreted as field access and outside the module it is interpreted as an accessor proc call.

12.4 Command invocation syntax

Routines can be invoked without the `()` if the call is syntactically a statement. This command invocation syntax also works for expressions, but then only a single argument may follow. This restriction means `echo f 1, f 2` is parsed as `echo(f(1), f(2))` and not as `echo(f(1, f(2)))`. The method call syntax may be used to provide one more argument in this case:

```
proc optarg(x: int, y: int = 0): int = x + y
proc singlearg(x: int): int = 20*x

echo optarg 1, " ", singlearg 2  # prints "1 40"

let fail = optarg 1, optarg 8   # Wrong. Too many arguments for a command call
let x = optarg(1, optarg 8)  # traditional procedure call with 2 arguments
let y = 1.optarg optarg 8    # same thing as above, w/o the parenthesis
assert x == y
```

The command invocation syntax also can't have complex expressions as arguments. For example: (anonymous procs), `if`, `case` or `try`. Function calls with no arguments still needs () to distinguish between a call and the function itself as a first class value.

12.5 Closures

Procedures can appear at the top level in a module as well as inside other scopes, in which case they are called nested procs. A nested proc can access local variables from its enclosing scope and if it does so it becomes a closure. Any captured variables are stored in a hidden additional argument to the closure (its environment) and they are accessed by reference by both the closure and its enclosing scope (i.e. any modifications made to them are visible in both places). The closure environment may be allocated on the heap or on the stack if the compiler determines that this would be safe.

12.5.1 Creating closures in loops

Since closures capture local variables by reference it is often not wanted behavior inside loop bodies. See closureScope and capture for details on how to change this behavior.

12.6 Anonymous Procs

Unnamed procedures can be used as lambda expressions to pass into other procedures:

```
var cities = @["Frankfurt", "Tokyo", "New York", "Kyiv"]

cities.sort(proc (x,y: string): int =
    cmp(x.len, y.len))
```

Procs as expressions can appear both as nested procs and inside top level executable code. The sugar module contains the => macro which enables a more succinct syntax for anonymous procedures resembling lambdas as they are in languages like JavaScript, C#, etc.

12.7 Func

The `func` keyword introduces a shortcut for a noSideEffect proc.

```
func binarySearch[T](a: openArray[T]; elem: T): int
```

Is short for:

```
proc binarySearch[T](a: openArray[T]; elem: T): int {.noSideEffect.}
```

12.8 Nonoverloadable builtins

The following builtin procs cannot be overloaded for reasons of implementation simplicity (they require specialized semantic checking):

```
declared, defined, definedInScope, compiles, sizeof,
is, shallowCopy, getAst, astToStr, spawn, procCall
```

Thus they act more like keywords than like ordinary identifiers; unlike a keyword however, a redefinition may shadow the definition in the `system` module. From this list the following should not be written in dot notation `x.f` since `x` cannot be type checked before it gets passed to `f`:

```
declared, defined, definedInScope, compiles, getAst, astToStr
```

12.9 Var parameters

The type of a parameter may be prefixed with the `var` keyword:

```
proc divmod(a, b: int; res, remainder: var int) =
  res = a div b
  remainder = a mod b

var
  x, y: int

divmod(8, 5, x, y) # modifies x and y
assert x == 1
assert y == 3
```

In the example, `res` and `remainder` are `var parameters`. Var parameters can be modified by the procedure and the changes are visible to the caller. The argument passed to a var parameter has to be an l-value. Var parameters are implemented as hidden pointers. The above example is equivalent to:

```
proc divmod(a, b: int; res, remainder: ptr int) =
  res[] = a div b
  remainder[] = a mod b

var
  x, y: int
divmod(8, 5, addr(x), addr(y))
assert x == 1
assert y == 3
```

In the examples, var parameters or pointers are used to provide two return values. This can be done in a cleaner way by returning a tuple:

```
proc divmod(a, b: int): tuple[res, remainder: int] =
  (a div b, a mod b)

var t = divmod(8, 5)

assert t.res == 1
assert t.remainder == 3
```

One can use tuple unpacking to access the tuple's fields:

```
var (x, y) = divmod(8, 5) # tuple unpacking
assert x == 1
assert y == 3
```

Note: `var` parameters are never necessary for efficient parameter passing. Since non-var parameters cannot be modified the compiler is always free to pass arguments by reference if it considers it can speed up execution.

12.10 Var return type

A proc, converter or iterator may return a `var` type which means that the returned value is an l-value and can be modified by the caller:

```
var g = 0

proc writeAccessToG(): var int =
  result = g

writeAccessToG() = 6
assert g == 6
```

It is a static error if the implicitly introduced pointer could be used to access a location beyond its lifetime:

```
proc writeAccessToG(): var int =
  var g = 0
  result = g # Error!
```

For iterators, a component of a tuple return type can have a `var` type too:

```
iterator mpairs(a: var seq[string]): tuple[key: int, val: var string] =
  for i in 0..a.high:
   yield (i, a[i])
```

In the standard library every name of a routine that returns a `var` type starts with the prefix `m` per convention.

Memory safety for returning by `var T` is ensured by a simple borrowing rule: If `result` does not refer to a location pointing to the heap (that is in `result = X` the `X` involves a `ptr` or `ref` access) then it has to be derived from the routine's first parameter:

```nim
proc forward[T](x: var T): var T =
  result = x # ok, derived from the first parameter.

proc p(param: var int): var int =
  var x: int
  # we know 'forward' provides a view into the location derived from
  # its first argument 'x'.
  result = forward(x) # Error: location is derived from ``x``
                      # which is not p's first parameter and lives
                      # on the stack.
```

In other words, the lifetime of what `result` points to is attached to the lifetime of the first parameter and that is enough knowledge to verify memory safety at the callsite.

12.10.1 Future directions

Later versions of Nim can be more precise about the borrowing rule with a syntax like:

```nim
proc foo(other: Y; container: var X): var T from container
```

Here `var T from container` explicitly exposes that the location is derived from the second parameter (called 'container' in this case). The syntax `var T from p` specifies a type `varTy[T, 2]` which is incompatible with `varTy[T, 1]`.

12.11 NRVO

Note: This section describes the current implementation. This part of the language specification will be changed. See https://github.com/nim-lang/RFCs/issues/230 for more information.

The return value is represented inside the body of a routine as the special result variable. This allows for a mechanism much like C++'s "named return value optimization" (NRVO). NRVO means that the stores to `result` inside `p` directly affect the destination `dest` in `let/var dest = p(args)` (definition of `dest`) and also in `dest = p(args)` (assignment to `dest`). This is achieved by rewriting `dest = p(args)` to `p'(args, dest)` where `p'` is a variation of `p` that returns `void` and receives a hidden mutable parameter representing `result`.

Informally:

```nim
proc p(): BigT = ...

var x = p()
x = p()

# is roughly turned into:

proc p(result: var BigT) = ...

var x; p(x)
p(x)
```

Let `T`'s be `p`'s return type. NRVO applies for `T` if `sizeof(T) >= N` (where `N` is implementation dependent), in other words, it applies for "big" structures.

If `p` can raise an exception, NRVO applies regardless. This can produce observable differences in behavior:

```nim
type
  BigT = array[16, int]

proc p(raiseAt: int): BigT =
  for i in 0..high(result):
    if i == raiseAt: raise newException(ValueError, "interception")
    result[i] = i

proc main =
  var x: BigT
  try:
    x = p(8)
```

```
  except ValueError:
    doAssert x == [0, 1, 2, 3, 4, 5, 6, 7, 0, 0, 0, 0, 0, 0, 0, 0]

main()
```

However, the current implementation produces a warning in these cases. There are different ways to deal with this warning:

1. Disable the warning via `{.push warning[ObservableStores]: off.}` ... `{.pop.}`. Then one may need to ensure that `p` only raises *before* any stores to `result` happen.
2. One can use a temporary helper variable, for example instead of `x = p(8)` use `let tmp = p(8); x = tmp`.

12.12 Overloading of the subscript operator

The `[]` subscript operator for arrays/openarrays/sequences can be overloaded.

13 Multi-methods

Note: Starting from Nim 0.20, to use multi-methods one must explicitly pass `–multimethods:on` when compiling.

Procedures always use static dispatch. Multi-methods use dynamic dispatch. For dynamic dispatch to work on an object it should be a reference type.

```
type
  Expression = ref object of RootObj ## abstract base class for an expression
  Literal = ref object of Expression
    x: int
  PlusExpr = ref object of Expression
    a, b: Expression

method eval(e: Expression): int {.base.} =
  # override this base method
  raise newException(CatchableError, "Method without implementation override")

method eval(e: Literal): int = return e.x

method eval(e: PlusExpr): int =
  # watch out: relies on dynamic binding
  result = eval(e.a) + eval(e.b)

proc newLit(x: int): Literal =
  new(result)
  result.x = x

proc newPlus(a, b: Expression): PlusExpr =
  new(result)
  result.a = a
  result.b = b

echo eval(newPlus(newPlus(newLit(1), newLit(2)), newLit(4)))
```

In the example the constructors `newLit` and `newPlus` are procs because they should use static binding, but `eval` is a method because it requires dynamic binding.

As can be seen in the example, base methods have to be annotated with the base pragma. The `base` pragma also acts as a reminder for the programmer that a base method `m` is used as the foundation to determine all the effects that a call to `m` might cause.

Note: Compile-time execution is not (yet) supported for methods.

Note: Starting from Nim 0.20, generic methods are deprecated.

13.1 Inhibit dynamic method resolution via procCall

Dynamic method resolution can be inhibited via the builtin system.procCall. This is somewhat comparable to the super keyword that traditional OOP languages offer.

```
type
  Thing = ref object of RootObj
  Unit = ref object of Thing
    x: int

method m(a: Thing) {.base.} =
  echo "base"

method m(a: Unit) =
  # Call the base method:
  procCall m(Thing(a))
  echo "1"
```

14 Iterators and the for statement

The for statement is an abstract mechanism to iterate over the elements of a container. It relies on an iterator to do so. Like `while` statements, `for` statements open an implicit block, so that they can be left with a `break` statement.

The `for` loop declares iteration variables - their scope reaches until the end of the loop body. The iteration variables' types are inferred by the return type of the iterator.

An iterator is similar to a procedure, except that it can be called in the context of a `for` loop. Iterators provide a way to specify the iteration over an abstract type. A key role in the execution of a `for` loop plays the `yield` statement in the called iterator. Whenever a `yield` statement is reached the data is bound to the `for` loop variables and control continues in the body of the `for` loop. The iterator's local variables and execution state are automatically saved between calls. Example:

```
# this definition exists in the system module
iterator items*(a: string): char {.inline.} =
  var i = 0
  while i < len(a):
    yield a[i]
    inc(i)

for ch in items("hello world"): # 'ch' is an iteration variable
  echo ch
```

The compiler generates code as if the programmer would have written this:

```
var i = 0
while i < len(a):
  var ch = a[i]
  echo ch
  inc(i)
```

If the iterator yields a tuple, there can be as many iteration variables as there are components in the tuple. The i'th iteration variable's type is the type of the i'th component. In other words, implicit tuple unpacking in a for loop context is supported.

14.1 Implicit items/pairs invocations

If the for loop expression `e` does not denote an iterator and the for loop has exactly 1 variable, the for loop expression is rewritten to `items(e)`; ie. an `items` iterator is implicitly invoked:

```
for x in [1,2,3]: echo x
```

If the for loop has exactly 2 variables, a `pairs` iterator is implicitly invoked.

Symbol lookup of the identifiers `items`/`pairs` is performed after the rewriting step, so that all overloads of `items`/`pairs` are taken into account.

14.2 First class iterators

There are 2 kinds of iterators in Nim: *inline* and *closure* iterators. An inline iterator is an iterator that's always inlined by the compiler leading to zero overhead for the abstraction, but may result in a heavy increase in code size.

Caution: the body of a for loop over an inline iterator is inlined into each `yield` statement appearing in the iterator code, so ideally the code should be refactored to contain a single yield when possible to avoid code bloat.

Inline iterators are second class citizens; They can be passed as parameters only to other inlining code facilities like templates, macros and other inline iterators.

In contrast to that, a closure iterator can be passed around more freely:

```
iterator count0(): int {.closure.} =
  yield 0

iterator count2(): int {.closure.} =
  var x = 1
  yield x
  inc x
  yield x

proc invoke(iter: iterator(): int {.closure.}) =
  for x in iter(): echo x

invoke(count0)
invoke(count2)
```

Closure iterators and inline iterators have some restrictions:

1. For now, a closure iterator cannot be executed at compile time.
2. `return` is allowed in a closure iterator but not in an inline iterator (but rarely useful) and ends the iteration.
3. Neither inline nor closure iterators can be recursive.
4. Neither inline nor closure iterators have the special `result` variable.
5. Closure iterators are not supported by the js backend.

Iterators that are neither marked `{.closure.}` nor `{.inline.}` explicitly default to being inline, but this may change in future versions of the implementation.

The `iterator` type is always of the calling convention `closure` implicitly; the following example shows how to use iterators to implement a collaborative tasking system:

```
# simple tasking:
type
  Task = iterator (ticker: int)

iterator a1(ticker: int) {.closure.} =
  echo "a1: A"
  yield
  echo "a1: B"
  yield
  echo "a1: C"
  yield
  echo "a1: D"

iterator a2(ticker: int) {.closure.} =
  echo "a2: A"
  yield
  echo "a2: B"
  yield
  echo "a2: C"

proc runTasks(t: varargs[Task]) =
```

```
  var ticker = 0
  while true:
    let x = t[ticker mod t.len]
    if finished(x): break
    x(ticker)
    inc ticker

runTasks(a1, a2)
```

The builtin `system.finished` can be used to determine if an iterator has finished its operation; no exception is raised on an attempt to invoke an iterator that has already finished its work.

Note that `system.finished` is error prone to use because it only returns `true` one iteration after the iterator has finished:

```
iterator mycount(a, b: int): int {.closure.} =
  var x = a
  while x <= b:
    yield x
    inc x

var c = mycount # instantiate the iterator
while not finished(c):
  echo c(1, 3)

# Produces
1
2
3
0
```

Instead this code has to be used:

```
var c = mycount # instantiate the iterator
while true:
  let value = c(1, 3)
  if finished(c): break # and discard 'value'!
  echo value
```

It helps to think that the iterator actually returns a pair `(value, done)` and `finished` is used to access the hidden `done` field.

Closure iterators are *resumable functions* and so one has to provide the arguments to every call. To get around this limitation one can capture parameters of an outer factory proc:

```
proc mycount(a, b: int): iterator (): int =
  result = iterator (): int =
    var x = a
    while x <= b:
      yield x
      inc x

let foo = mycount(1, 4)

for f in foo():
  echo f
```

15 Converters

A converter is like an ordinary proc except that it enhances the "implicitly convertible" type relation (see Convertible relation):

```
# bad style ahead: Nim is not C.
converter toBool(x: int): bool = x != 0

if 4:
  echo "compiles"
```

A converter can also be explicitly invoked for improved readability. Note that implicit converter chaining is not supported: If there is a converter from type A to type B and from type B to type C the implicit conversion from A to C is not provided.

16 Type sections

Example:

```
type # example demonstrating mutually recursive types
  Node = ref object  # an object managed by the garbage collector (ref)
    le, ri: Node     # left and right subtrees
    sym: ref Sym     # leaves contain a reference to a Sym

  Sym = object       # a symbol
    name: string     # the symbol's name
    line: int        # the line the symbol was declared in
    code: Node       # the symbol's abstract syntax tree
```

A type section begins with the `type` keyword. It contains multiple type definitions. A type definition binds a type to a name. Type definitions can be recursive or even mutually recursive. Mutually recursive types are only possible within a single `type` section. Nominal types like `objects` or `enums` can only be defined in a `type` section.

17 Exception handling

17.1 Try statement

Example:

```
# read the first two lines of a text file that should contain numbers
# and tries to add them
var
  f: File
if open(f, "numbers.txt"):
  try:
    var a = readLine(f)
    var b = readLine(f)
    echo "sum: " & $(parseInt(a) + parseInt(b))
  except OverflowDefect:
    echo "overflow!"
  except ValueError:
    echo "could not convert string to integer"
  except IOError:
    echo "IO error!"
  except:
    echo "Unknown exception!"
  finally:
    close(f)
```

The statements after the `try` are executed in sequential order unless an exception `e` is raised. If the exception type of `e` matches any listed in an `except` clause the corresponding statements are executed. The statements following the `except` clauses are called exception handlers.

The empty except clause is executed if there is an exception that is not listed otherwise. It is similar to an `else` clause in `if` statements.

If there is a finally clause, it is always executed after the exception handlers.

The exception is *consumed* in an exception handler. However, an exception handler may raise another exception. If the exception is not handled, it is propagated through the call stack. This means that often the rest of the procedure - that is not within a `finally` clause - is not executed (if an exception occurs).

17.2 Try expression

Try can also be used as an expression; the type of the `try` branch then needs to fit the types of `except` branches, but the type of the `finally` branch always has to be `void`:

```
from strutils import parseInt

let x = try: parseInt("133a")
        except: -1
        finally: echo "hi"
```

To prevent confusing code there is a parsing limitation; if the `try` follows a `(` it has to be written as a one liner:

```
let x = (try: parseInt("133a") except: -1)
```

17.3 Except clauses

Within an `except` clause it is possible to access the current exception using the following syntax:

```
try:
  # ...
except IOError as e:
  # Now use "e"
  echo "I/O error: " & e.msg
```

Alternatively, it is possible to use `getCurrentException` to retrieve the exception that has been raised:

```
try:
  # ...
except IOError:
  let e = getCurrentException()
  # Now use "e"
```

Note that `getCurrentException` always returns a `ref Exception` type. If a variable of the proper type is needed (in the example above, `IOError`), one must convert it explicitly:

```
try:
  # ...
except IOError:
  let e = (ref IOError)(getCurrentException())
  # "e" is now of the proper type
```

However, this is seldom needed. The most common case is to extract an error message from `e`, and for such situations it is enough to use `getCurrentExceptionMsg`:

```
try:
  # ...
except:
  echo getCurrentExceptionMsg()
```

17.4 Custom exceptions

Is it possible to create custom exceptions. A custom exception is a custom type:

```
type
  LoadError* = object of Exception
```

Ending the custom exception's name with `Error` is recommended.

Custom exceptions can be raised like any others, e.g.:

```
raise newException(LoadError, "Failed to load data")
```

17.5 Defer statement

Instead of a `try finally` statement a `defer` statement can be used.

Any statements following the `defer` in the current block will be considered to be in an implicit try block:

```
proc main =
  var f = open("numbers.txt")
  defer: close(f)
  f.write "abc"
  f.write "def"
```

Is rewritten to:

```
proc main =
  var f = open("numbers.txt")
  try:
    f.write "abc"
    f.write "def"
  finally:
    close(f)
```

Top level `defer` statements are not supported since it's unclear what such a statement should refer to.

17.6 Raise statement

Example:

```
raise newException(IOError, "IO failed")
```

Apart from built-in operations like array indexing, memory allocation, etc. the `raise` statement is the only way to raise an exception.

If no exception name is given, the current exception is re-raised. The ReraiseDefect exception is raised if there is no exception to re-raise. It follows that the `raise` statement *always* raises an exception.

17.7 Exception hierarchy

The exception tree is defined in the system module. Every exception inherits from `system.Exception`. Exceptions that indicate programming bugs inherit from `system.Defect` (which is a subtype of `Exception`) and are strictly speaking not catchable as they can also be mapped to an operation that terminates the whole process. If panics are turned into exceptions, these exceptions inherit from `Defect`.

Exceptions that indicate any other runtime error that can be caught inherit from `system.CatchableError` (which is a subtype of `Exception`).

17.8 Imported exceptions

It is possible to raise/catch imported C++ exceptions. Types imported using `importcpp` can be raised or caught. Exceptions are raised by value and caught by reference. Example:

```
type
  CStdException {.importcpp: "std::exception", header: "<exception>", inheritable.} = object
    ## does not inherit from `RootObj`, so we use `inheritable` instead
  CRuntimeError {.requiresInit, importcpp: "std::runtime_error", header: "<stdexcept>".} = object of CStdExceptic
    ## `CRuntimeError` has no default constructor => `requiresInit`
proc what(s: CStdException): cstring {.importcpp: "((char *)#.what())".}
proc initRuntimeError(a: cstring): CRuntimeError {.importcpp: "std::runtime_error(@)", constructor.}
proc initStdException(): CStdException {.importcpp: "std::exception()", constructor.}

proc fn() =
  let a = initRuntimeError("foo")
  doAssert $a.what == "foo"
  var b: cstring
  try: raise initRuntimeError("foo2")
  except CStdException as e:
    doAssert e is CStdException
    b = e.what()
  doAssert $b == "foo2"

  try: raise initStdException()
  except CStdException: discard

  try: raise initRuntimeError("foo3")
  except CRuntimeError as e:
    b = e.what()
  except CStdException:
    doAssert false
```

```
  doAssert $b == "foo3"

fn()
```

Note: `getCurrentException()` and `getCurrentExceptionMsg()` are not available for imported exceptions from C++. One needs to use the `except ImportedException as x:` syntax and rely on functionality of the `x` object to get exception details.

18 Effect system

18.1 Exception tracking

Nim supports exception tracking. The raises pragma can be used to explicitly define which exceptions a proc/iterator/method/converter is allowed to raise. The compiler verifies this:

```
proc p(what: bool) {.raises: [IOError, OSError].} =
  if what: raise newException(IOError, "IO")
  else: raise newException(OSError, "OS")
```

An empty `raises` list (`raises: []`) means that no exception may be raised:

```
proc p(): bool {.raises: [].} =
  try:
    unsafeCall()
    result = true
  except:
    result = false
```

A `raises` list can also be attached to a proc type. This affects type compatibility:

```
type
  Callback = proc (s: string) {.raises: [IOError].}
var
  c: Callback

proc p(x: string) =
  raise newException(OSError, "OS")

c = p # type error
```

For a routine `p` the compiler uses inference rules to determine the set of possibly raised exceptions; the algorithm operates on `p`'s call graph:

1. Every indirect call via some proc type `T` is assumed to raise `system.Exception` (the base type of the exception hierarchy) and thus any exception unless `T` has an explicit `raises` list. However if the call is of the form `f(...)` where `f` is a parameter of the currently analysed routine it is ignored. The call is optimistically assumed to have no effect. Rule 2 compensates for this case.

2. Every expression of some proc type within a call that is not a call itself (and not nil) is assumed to be called indirectly somehow and thus its raises list is added to `p`'s raises list.

3. Every call to a proc `q` which has an unknown body (due to a forward declaration or an `importc` pragma) is assumed to raise `system.Exception` unless `q` has an explicit `raises` list.

4. Every call to a method `m` is assumed to raise `system.Exception` unless `m` has an explicit `raises` list.

5. For every other call the analysis can determine an exact `raises` list.

6. For determining a `raises` list, the `raise` and `try` statements of `p` are taken into consideration.

Rules 1-2 ensure the following works:

```
proc noRaise(x: proc()) {.raises: [].} =
  # unknown call that might raise anything, but valid:
  x()

proc doRaise() {.raises: [IOError].} =
  raise newException(IOError, "IO")

proc use() {.raises: [].} =
  # doesn't compile! Can raise IOError!
  noRaise(doRaise)
```

So in many cases a callback does not cause the compiler to be overly conservative in its effect analysis.

Exceptions inheriting from `system.Defect` are not tracked with the `.raises: []` exception tracking mechanism. This is more consistent with the built-in operations. The following code is valid::

```
proc mydiv(a, b): int {.raises: [].} =
  a div b # can raise an DivByZeroDefect
```

And so is::

```
proc mydiv(a, b): int {.raises: [].} =
  if b == 0: raise newException(DivByZeroDefect, "division by zero")
  else: result = a div b
```

The reason for this is that `DivByZeroDefect` inherits from `Defect` and with `-panics:on` Defects become unrecoverable errors. (Since version 1.4 of the language.)

18.2 Tag tracking

The exception tracking is part of Nim's effect system. Raising an exception is an *effect*. Other effects can also be defined. A user defined effect is a means to *tag* a routine and to perform checks against this tag:

```
type IO = object ## input/output effect
proc readLine(): string {.tags: [IO].} = discard

proc no_IO_please() {.tags: [].} =
  # the compiler prevents this:
  let x = readLine()
```

A tag has to be a type name. A `tags` list - like a `raises` list - can also be attached to a proc type. This affects type compatibility.

The inference for tag tracking is analogous to the inference for exception tracking.

18.3 Effects pragma

The `effects` pragma has been designed to assist the programmer with the effects analysis. It is a statement that makes the compiler output all inferred effects up to the `effects`'s position:

```
proc p(what: bool) =
  if what:
    raise newException(IOError, "IO")
    {.effects.}
  else:
    raise newException(OSError, "OS")
```

The compiler produces a hint message that `IOError` can be raised. `OSError` is not listed as it cannot be raised in the branch the `effects` pragma appears in.

19 Generics

Generics are Nim's means to parametrize procs, iterators or types with type parameters. Depending on context, the brackets are used either to introduce type parameters or to instantiate a generic proc, iterator or type.

The following example shows a generic binary tree can be modelled:

```
type
  BinaryTree*[T] = ref object # BinaryTree is a generic type with
                              # generic param ''T''
    le, ri: BinaryTree[T]     # left and right subtrees; may be nil
    data: T                   # the data stored in a node

proc newNode*[T](data: T): BinaryTree[T] =
  # constructor for a node
  result = BinaryTree[T](le: nil, ri: nil, data: data)

proc add*[T](root: var BinaryTree[T], n: BinaryTree[T]) =
  # insert a node into the tree
  if root == nil:
    root = n
  else:
    var it = root
    while it != nil:
      # compare the data items; uses the generic ''cmp'' proc
      # that works for any type that has a ''=='' and ''<'' operator
      var c = cmp(it.data, n.data)
      if c < 0:
        if it.le == nil:
          it.le = n
          return
        it = it.le
      else:
        if it.ri == nil:
          it.ri = n
          return
        it = it.ri

proc add*[T](root: var BinaryTree[T], data: T) =
  # convenience proc:
  add(root, newNode(data))

iterator preorder*[T](root: BinaryTree[T]): T =
  # Preorder traversal of a binary tree.
  # Since recursive iterators are not yet implemented,
  # this uses an explicit stack (which is more efficient anyway):
  var stack: seq[BinaryTree[T]] = @[root]
  while stack.len > 0:
    var n = stack.pop()
    while n != nil:
      yield n.data
      add(stack, n.ri)  # push right subtree onto the stack
      n = n.le          # and follow the left pointer

var
  root: BinaryTree[string] # instantiate a BinaryTree with ''string''
add(root, newNode("hello")) # instantiates ''newNode'' and ''add''
add(root, "world")          # instantiates the second ''add'' proc
for str in preorder(root):
  stdout.writeLine(str)
```

The `T` is called a generic type parameter or a type variable.

19.1 Is operator

The `is` operator is evaluated during semantic analysis to check for type equivalence. It is therefore very useful for type specialization within generic code:

```
type
  Table[Key, Value] = object
    keys: seq[Key]
    values: seq[Value]
    when not (Key is string): # empty value for strings used for optimization
      deletedKeys: seq[bool]
```

type class	matches
`object`	any object type
`tuple`	any tuple type
`enum`	any enumeration
`proc`	any proc type
`ref`	any `ref` type
`ptr`	any `ptr` type
`var`	any `var` type
`distinct`	any distinct type
`array`	any array type
`set`	any set type
`seq`	any seq type
`auto`	any type
`any`	distinct auto (see below)

19.2 Type Classes

A type class is a special pseudo-type that can be used to match against types in the context of overload resolution or the `is` operator. Nim supports the following built-in type classes:

Furthermore, every generic type automatically creates a type class of the same name that will match any instantiation of the generic type.

Type classes can be combined using the standard boolean operators to form more complex type classes:

```
# create a type class that will match all tuple and object types
type RecordType = tuple or object

proc printFields[T: RecordType](rec: T) =
  for key, value in fieldPairs(rec):
    echo key, " = ", value
```

Whilst the syntax of type classes appears to resemble that of ADTs/algebraic data types in ML-like languages, it should be understood that type classes are static constraints to be enforced at type instantiations. Type classes are not really types in themselves, but are instead a system of providing generic "checks" that ultimately *resolve* to some singular type. Type classes do not allow for runtime type dynamism, unlike object variants or methods.

As an example, the following would not compile:

```
type TypeClass = int | string
var foo: TypeClass = 2 # foo’s type is resolved to an int here
foo = "this will fail" # error here, because foo is an int
```

Nim allows for type classes and regular types to be specified as type constraints of the generic type parameter:

```
proc onlyIntOrString[T: int|string](x, y: T) = discard

onlyIntOrString(450, 616) # valid
onlyIntOrString(5.0, 0.0) # type mismatch
onlyIntOrString("xy", 50) # invalid as ’T’ cannot be both at the same time
```

19.3 Implicit generics

A type class can be used directly as the parameter’s type.

```
# create a type class that will match all tuple and object types
type RecordType = tuple or object

proc printFields(rec: RecordType) =
  for key, value in fieldPairs(rec):
    echo key, " = ", value
```

Procedures utilizing type classes in such manner are considered to be implicitly generic. They will be instantiated once for each unique combination of param types used within the program.

By default, during overload resolution each named type class will bind to exactly one concrete type. We call such type classes bind once types. Here is an example taken directly from the system module to illustrate this:

```
proc `==`*(x, y: tuple): bool =
  ## requires `x` and `y` to be of the same tuple type
  ## generic ``==`` operator for tuples that is lifted from the components
  ## of `x` and `y`.
  result = true
  for a, b in fields(x, y):
    if a != b: result = false
```

Alternatively, the `distinct` type modifier can be applied to the type class to allow each param matching the type class to bind to a different type. Such type classes are called bind many types.

Procs written with the implicitly generic style will often need to refer to the type parameters of the matched generic type. They can be easily accessed using the dot syntax:

```
type Matrix[T, Rows, Columns] = object
  ...

proc `[]`(m: Matrix, row, col: int): Matrix.T =
  m.data[col * high(Matrix.Columns) + row]
```

Here are more examples that illustrate implicit generics:

```
proc p(t: Table; k: Table.Key): Table.Value

# is roughly the same as:

proc p[Key, Value](t: Table[Key, Value]; k: Key): Value
```

```
proc p(a: Table, b: Table)

# is roughly the same as:

proc p[Key, Value](a, b: Table[Key, Value])
```

```
proc p(a: Table, b: distinct Table)

# is roughly the same as:

proc p[Key, Value, KeyB, ValueB](a: Table[Key, Value], b: Table[KeyB, ValueB])
```

`typedesc` used as a parameter type also introduces an implicit generic. `typedesc` has its own set of rules:

```
proc p(a: typedesc)

# is roughly the same as:

proc p[T](a: typedesc[T])
```

`typedesc` is a "bind many" type class:

```
proc p(a, b: typedesc)

# is roughly the same as:

proc p[T, T2](a: typedesc[T], b: typedesc[T2])
```

A parameter of type `typedesc` is itself usable as a type. If it is used as a type, it's the underlying type. (In other words, one level of "typedesc"-ness is stripped off:

```
proc p(a: typedesc; b: a) = discard

# is roughly the same as:
proc p[T](a: typedesc[T]; b: T) = discard

# hence this is a valid call:
p(int, 4)
# as parameter 'a' requires a type, but 'b' requires a value.
```

19.4 Generic inference restrictions

The types `var T`, `out T` and `typedesc[T]` cannot be inferred in a generic instantiation. The following is not allowed:

```
proc g[T](f: proc(x: T); x: T) =
  f(x)

proc c(y: int) = echo y
proc v(y: var int) =
  y += 100
var i: int

# allowed: infers 'T' to be of type 'int'
g(c, 42)

# not valid: 'T' is not inferred to be of type 'var int'
g(v, i)

# also not allowed: explicit instantiation via 'var int'
g[var int](v, i)
```

19.5 Symbol lookup in generics

19.5.1 Open and Closed symbols

The symbol binding rules in generics are slightly subtle: There are "open" and "closed" symbols. A "closed" symbol cannot be re-bound in the instantiation context, an "open" symbol can. Per default overloaded symbols are open and every other symbol is closed.

Open symbols are looked up in two different contexts: Both the context at definition and the context at instantiation are considered:

```
type
  Index = distinct int

proc `==` (a, b: Index): bool {.borrow.}

var a = (0, 0.Index)
var b = (0, 0.Index)

echo a == b # works!
```

In the example the generic == for tuples (as defined in the system module) uses the == operators of the tuple's components. However, the == for the `Index` type is defined *after* the == for tuples; yet the example compiles as the instantiation takes the currently defined symbols into account too.

19.6 Mixin statement

A symbol can be forced to be open by a mixin declaration:

```
proc create*[T](): ref T =
  # there is no overloaded 'init' here, so we need to state that it's an
  # open symbol explicitly:
  mixin init
  new result
  init result
```

`mixin` statements only make sense in templates and generics.

19.7 Bind statement

The `bind` statement is the counterpart to the `mixin` statement. It can be used to explicitly declare identifiers that should be bound early (i.e. the identifiers should be looked up in the scope of the template/generic definition):

```
# Module A
var
  lastId = 0

template genId*: untyped =
  bind lastId
  inc(lastId)
  lastId

# Module B
import A

echo genId()
```

But a `bind` is rarely useful because symbol binding from the definition scope is the default.
`bind` statements only make sense in templates and generics.

20 Templates

A template is a simple form of a macro: It is a simple substitution mechanism that operates on Nim's abstract syntax trees. It is processed in the semantic pass of the compiler.

The syntax to *invoke* a template is the same as calling a procedure.

Example:

```
template `!=` (a, b: untyped): untyped =
  # this definition exists in the System module
  not (a == b)

assert(5 != 6) # the compiler rewrites that to: assert(not (5 == 6))
```

The `!=`, `>`, `>=`, `in`, `notin`, `isnot` operators are in fact templates:
`a > b` is transformed into `b < a`.
`a in b` is transformed into `contains(b, a)`.
`notin` and `isnot` have the obvious meanings.

The "types" of templates can be the symbols `untyped`, `typed` or `typedesc`. These are "meta types", they can only be used in certain contexts. Regular types can be used too; this implies that `typed` expressions are expected.

20.1 Typed vs untyped parameters

An `untyped` parameter means that symbol lookups and type resolution is not performed before the expression is passed to the template. This means that for example *undeclared* identifiers can be passed to the template:

```
template declareInt(x: untyped) =
  var x: int

declareInt(x) # valid
x = 3

template declareInt(x: typed) =
  var x: int

declareInt(x) # invalid, because x has not been declared and so has no type
```

A template where every parameter is `untyped` is called an immediate template. For historical reasons templates can be explicitly annotated with an `immediate` pragma and then these templates do not take part in overloading resolution and the parameters' types are *ignored* by the compiler. Explicit immediate templates are now deprecated.

Note: For historical reasons `stmt` was an alias for `typed` and `expr` was an alias for `untyped`, but they are removed.

20.2 Passing a code block to a template

One can pass a block of statements as the last argument to a template following the special **:** syntax:

```
template withFile(f, fn, mode, actions: untyped): untyped =
  var f: File
  if open(f, fn, mode):
    try:
      actions
    finally:
      close(f)
  else:
    quit("cannot open: " & fn)

withFile(txt, "ttempl3.txt", fmWrite):  # special colon
  txt.writeLine("line 1")
  txt.writeLine("line 2")
```

In the example, the two `writeLine` statements are bound to the `actions` parameter.

Usually to pass a block of code to a template the parameter that accepts the block needs to be of type `untyped`. Because symbol lookups are then delayed until template instantiation time:

```
template t(body: typed) =
  block:
    body

t:
  var i = 1
  echo i

t:
  var i = 2  # fails with 'attempt to redeclare i'
  echo i
```

The above code fails with the mysterious error message that `i` has already been declared. The reason for this is that the `var i = ...` bodies need to be type-checked before they are passed to the `body` parameter and type checking in Nim implies symbol lookups. For the symbol lookups to succeed `i` needs to be added to the current (i.e. outer) scope. After type checking these additions to the symbol table are not rolled back (for better or worse). The same code works with `untyped` as the passed body is not required to be type-checked:

```
template t(body: untyped) =
  block:
    body

t:
  var i = 1
  echo i

t:
  var i = 2  # compiles
  echo i
```

20.3 Varargs of untyped

In addition to the `untyped` meta-type that prevents type checking there is also `varargs[untyped]` so that not even the number of parameters is fixed:

```
template hideIdentifiers(x: varargs[untyped]) = discard

hideIdentifiers(undeclared1, undeclared2)
```

However, since a template cannot iterate over varargs, this feature is generally much more useful for macros.

20.4 Symbol binding in templates

A template is a hygienic macro and so opens a new scope. Most symbols are bound from the definition scope of the template:

```
# Module A
var
  lastId = 0

template genId*: untyped =
  inc(lastId)
  lastId

# Module B
import A

echo genId() # Works as 'lastId' has been bound in 'genId's defining scope
```

As in generics symbol binding can be influenced via `mixin` or `bind` statements.

20.5 Identifier construction

In templates identifiers can be constructed with the backticks notation:

```
template typedef(name: untyped, typ: typedesc) =
  type
    `T name`* {.inject.} = typ
    `P name`* {.inject.} = ref `T name`

typedef(myint, int)
var x: PMyInt
```

In the example `name` is instantiated with `myint`, so ‘T name‘ becomes `Tmyint`.

20.6 Lookup rules for template parameters

A parameter `p` in a template is even substituted in the expression `x.p`. Thus template arguments can be used as field names and a global symbol can be shadowed by the same argument name even when fully qualified:

```
# module 'm'

type
  Lev = enum
    levA, levB

var abclev = levB

template tstLev(abclev: Lev) =
  echo abclev, " ", m.abclev

tstLev(levA)
# produces: 'levA levA'
```

But the global symbol can properly be captured by a `bind` statement:

```
# module 'm'

type
  Lev = enum
    levA, levB

var abclev = levB

template tstLev(abclev: Lev) =
  bind m.abclev
  echo abclev, " ", m.abclev

tstLev(levA)
# produces: 'levA levB'
```

20.7 Hygiene in templates

Per default templates are hygienic: Local identifiers declared in a template cannot be accessed in the instantiation context:

```
template newException*(exceptn: typedesc, message: string): untyped =
  var
    e: ref exceptn  # e is implicitly gensym'ed here
  new(e)
  e.msg = message
  e

# so this works:
let e = "message"
raise newException(IoError, e)
```

Whether a symbol that is declared in a template is exposed to the instantiation scope is controlled by the inject and gensym pragmas: gensym'ed symbols are not exposed but inject'ed are.

The default for symbols of entity `type`, `var`, `let` and `const` is `gensym` and for `proc`, `iterator`, `converter`, `template`, `macro` is `inject`. However, if the name of the entity is passed as a template parameter, it is an inject'ed symbol:

```
template withFile(f, fn, mode: untyped, actions: untyped): untyped =
  block:
    var f: File  # since 'f' is a template param, it's injected implicitly
    ...

withFile(txt, "ttempl3.txt", fmWrite):
  txt.writeLine("line 1")
  txt.writeLine("line 2")
```

The `inject` and `gensym` pragmas are second class annotations; they have no semantics outside of a template definition and cannot be abstracted over:

```
{.pragma myInject: inject.}

template t() =
  var x {.myInject.}: int # does NOT work
```

To get rid of hygiene in templates, one can use the dirty pragma for a template. `inject` and `gensym` have no effect in `dirty` templates.

`gensym`'ed symbols cannot be used as `field` in the `x.field` syntax. Nor can they be used in the `ObjectConstruction(field: value)` and `namedParameterCall(field = value)` syntactic constructs.

The reason for this is that code like

```
type
  T = object
    f: int

template tmp(x: T) =
  let f = 34
  echo x.f, T(f: 4)
```

should work as expected.

However, this means that the method call syntax is not available for `gensym`'ed symbols:

```
template tmp(x) =
  type
    T {.gensym.} = int

  echo x.T # invalid: instead use:  'echo T(x)'.

tmp(12)
```

Note: The Nim compiler prior to version 1 was more lenient about this requirement. Use the `-useVersion:0.19` switch for a transition period.

20.8 Limitations of the method call syntax

The expression `x` in `x.f` needs to be semantically checked (that means symbol lookup and type checking) before it can be decided that it needs to be rewritten to `f(x)`. Therefore the dot syntax has some limitations when it is used to invoke templates/macros:

```
template declareVar(name: untyped) =
  const name {.inject.} = 45

# Doesn't compile:
unknownIdentifier.declareVar
```

Another common example is this:

```
from sequtils import toSeq

iterator something: string =
  yield "Hello"
  yield "World"

var info = something().toSeq
```

The problem here is that the compiler already decided that `something()` as an iterator is not callable in this context before `toSeq` gets its chance to convert it into a sequence.

It is also not possible to use fully qualified identifiers with module symbol in method call syntax. The order in which the dot operator binds to symbols prohibits this.

```
import sequtils

var myItems = @[1,3,3,7]
let N1 = count(myItems, 3) # OK
let N2 = sequtils.count(myItems, 3) # fully qualified, OK
let N3 = myItems.count(3) # OK
let N4 = myItems.sequtils.count(3) # illegal, `myItems.sequtils` can't be resolved
```

This means that when for some reason a procedure needs a disambiguation through the module name, the call needs to be written in function call syntax.

21 Macros

A macro is a special function that is executed at compile time. Normally the input for a macro is an abstract syntax tree (AST) of the code that is passed to it. The macro can then do transformations on it and return the transformed AST. This can be used to add custom language features and implement domain specific languages.

Macro invocation is a case where semantic analysis does **not** entirely proceed top to bottom and left to right. Instead, semantic analysis happens at least twice:

- Semantic analysis recognizes and resolves the macro invocation.
- The compiler executes the macro body (which may invoke other procs).
- It replaces the AST of the macro invocation with the AST returned by the macro.
- It repeats semantic analysis of that region of the code.
- If the AST returned by the macro contains other macro invocations, this process iterates.

While macros enable advanced compile-time code transformations, they cannot change Nim's syntax.

21.1 Debug Example

The following example implements a powerful `debug` command that accepts a variable number of arguments:

```
# to work with Nim syntax trees, we need an API that is defined in the
# ``macros`` module:
import macros

macro debug(args: varargs[untyped]): untyped =
  # `args` is a collection of `NimNode` values that each contain the
  # AST for an argument of the macro. A macro always has to
  # return a `NimNode`. A node of kind `nnkStmtList` is suitable for
  # this use case.
  result = nnkStmtList.newTree()
  # iterate over any argument that is passed to this macro:
  for n in args:
    # add a call to the statement list that writes the expression;
    # `toStrLit` converts an AST to its string representation:
    result.add newCall("write", newIdentNode("stdout"), newLit(n.repr))
    # add a call to the statement list that writes ": "
    result.add newCall("write", newIdentNode("stdout"), newLit(": "))
    # add a call to the statement list that writes the expressions value:
    result.add newCall("writeLine", newIdentNode("stdout"), n)

var
  a: array[0..10, int]
  x = "some string"
a[0] = 42
a[1] = 45

debug(a[0], a[1], x)
```

The macro call expands to:

```
write(stdout, "a[0]")
write(stdout, ": ")
writeLine(stdout, a[0])

write(stdout, "a[1]")
write(stdout, ": ")
writeLine(stdout, a[1])

write(stdout, "x")
write(stdout, ": ")
writeLine(stdout, x)
```

Arguments that are passed to a `varargs` parameter are wrapped in an array constructor expression. This is why `debug` iterates over all of `n`'s children.

21.2 BindSym

The above `debug` macro relies on the fact that `write`, `writeLine` and `stdout` are declared in the system module and thus visible in the instantiating context. There is a way to use bound identifiers (aka symbols) instead of using unbound identifiers. The `bindSym` builtin can be used for that:

```
import macros

macro debug(n: varargs[typed]): untyped =
  result = newNimNode(nnkStmtList, n)
  for x in n:
    # we can bind symbols in scope via 'bindSym':
    add(result, newCall(bindSym"write", bindSym"stdout", toStrLit(x)))
    add(result, newCall(bindSym"write", bindSym"stdout", newStrLitNode(": ")))
    add(result, newCall(bindSym"writeLine", bindSym"stdout", x))

var
  a: array[0..10, int]
```

```
  x = "some string"
a[0] = 42
a[1] = 45

debug(a[0], a[1], x)
```

The macro call expands to:

```
write(stdout, "a[0]")
write(stdout, ": ")
writeLine(stdout, a[0])

write(stdout, "a[1]")
write(stdout, ": ")
writeLine(stdout, a[1])

write(stdout, "x")
write(stdout, ": ")
writeLine(stdout, x)
```

However, the symbols `write`, `writeLine` and `stdout` are already bound and are not looked up again. As the example shows, `bindSym` does work with overloaded symbols implicitly.

21.3 Case-Of Macro

In Nim it is possible to have a macro with the syntax of a *case-of* expression just with the difference that all of branches are passed to and processed by the macro implementation. It is then up the macro implementation to transform the *of-branches* into a valid Nim statement. The following example should show how this feature could be used for a lexical analyzer.

```
import macros

macro case_token(args: varargs[untyped]): untyped =
  echo args.treeRepr
  # creates a lexical analyzer from regular expressions
  # ... (implementation is an exercise for the reader ;-)
  discard

case_token: # this colon tells the parser it is a macro statement
of r"[A-Za-z_]+[A-Za-z_0-9]*":
  return tkIdentifier
of r"0-9+":
  return tkInteger
of r"[\+\-\*\?]+":
  return tkOperator
else:
  return tkUnknown
```

Style note: For code readability, it is the best idea to use the least powerful programming construct that still suffices. So the "check list" is:

1. Use an ordinary proc/iterator, if possible.
2. Else: Use a generic proc/iterator, if possible.
3. Else: Use a template, if possible.
4. Else: Use a macro.

22 Special Types

22.1 static[T]

As their name suggests, static parameters must be constant expressions:

```
proc precompiledRegex(pattern: static string): RegEx =
  var res {.global.} = re(pattern)
  return res

precompiledRegex("/d+") # Replaces the call with a precompiled
                        # regex, stored in a global variable

precompiledRegex(paramStr(1)) # Error, command-line options
                              # are not constant expressions
```

For the purposes of code generation, all static params are treated as generic params - the proc will be compiled separately for each unique supplied value (or combination of values).

Static params can also appear in the signatures of generic types:

```
type
  Matrix[M,N: static int; T: Number] = array[0..(M*N - 1), T]
    # Note how 'Number' is just a type constraint here, while
    # 'static int' requires us to supply an int value

  AffineTransform2D[T] = Matrix[3, 3, T]
  AffineTransform3D[T] = Matrix[4, 4, T]

var m1: AffineTransform3D[float]  # OK
var m2: AffineTransform2D[string] # Error, 'string' is not a 'Number'
```

Please note that `static T` is just a syntactic convenience for the underlying generic type `static[T]`. The type param can be omitted to obtain the type class of all constant expressions. A more specific type class can be created by instantiating `static` with another type class.

One can force an expression to be evaluated at compile time as a constant expression by coercing it to a corresponding `static` type:

```
import math

echo static(fac(5)), " ", static[bool](16.isPowerOfTwo)
```

The compiler will report any failure to evaluate the expression or a possible type mismatch error.

22.2 typedesc[T]

In many contexts, Nim allows to treat the names of types as regular values. These values exists only during the compilation phase, but since all values must have a type, `typedesc` is considered their special type.

`typedesc` acts like a generic type. For instance, the type of the symbol `int` is `typedesc[int]`. Just like with regular generic types, when the generic param is omitted, `typedesc` denotes the type class of all types. As a syntactic convenience, one can also use `typedesc` as a modifier.

Procs featuring `typedesc` params are considered implicitly generic. They will be instantiated for each unique combination of supplied types and within the body of the proc, the name of each param will refer to the bound concrete type:

```
proc new(T: typedesc): ref T =
  echo "allocating ", T.name
  new(result)

var n = Node.new
var tree = new(BinaryTree[int])
```

When multiple type params are present, they will bind freely to different types. To force a bind-once behavior one can use an explicit generic param:

```
proc acceptOnlyTypePairs[T, U](A, B: typedesc[T]; C, D: typedesc[U])
```

Once bound, type params can appear in the rest of the proc signature:

```
template declareVariableWithType(T: typedesc, value: T) =
  var x: T = value

declareVariableWithType int, 42
```

Overload resolution can be further influenced by constraining the set of types that will match the type param. This works in practice to attaching attributes to types via templates. The constraint can be a concrete type or a type class.

```
template maxval(T: typedesc[int]): int = high(int)
template maxval(T: typedesc[float]): float = Inf

var i = int.maxval
var f = float.maxval
when false:
  var s = string.maxval # error, maxval is not implemented for string

template isNumber(t: typedesc[object]): string = "Don't think so."
template isNumber(t: typedesc[SomeInteger]): string = "Yes!"
template isNumber(t: typedesc[SomeFloat]): string = "Maybe, could be NaN."

echo "is int a number? ", isNumber(int)
echo "is float a number? ", isNumber(float)
echo "is RootObj a number? ", isNumber(RootObj)
```

Passing `typedesc` almost identical, just with the differences that the macro is not instantiated generically. The type expression is simply passed as a `NimNode` to the macro, like everything else.

```
import macros

macro forwardType(arg: typedesc): typedesc =
  # ''arg'' is of type ''NimNode''
  let tmp: NimNode = arg
  result = tmp

var tmp: forwardType(int)
```

22.3 typeof operator

Note: `typeof(x)` can for historical reasons also be written as `type(x)` but `type(x)` is discouraged.

One can obtain the type of a given expression by constructing a `typeof` value from it (in many other languages this is known as the typeof operator):

```
var x = 0
var y: typeof(x) # y has type int
```

If `typeof` is used to determine the result type of a proc/iterator/converter call `c(X)` (where `X` stands for a possibly empty list of arguments), the interpretation where `c` is an iterator is preferred over the other interpretations, but this behavior can be changed by passing `typeOfProc` as the second argument to `typeof`:

```
iterator split(s: string): string = discard
proc split(s: string): seq[string] = discard

# since an iterator is the preferred interpretation, 'y' has the type ''string'':
assert typeof("a b c".split) is string

assert typeof("a b c".split, typeOfProc) is seq[string]
```

23 Modules

Nim supports splitting a program into pieces by a module concept. Each module needs to be in its own file and has its own namespace. Modules enable information hiding and separate compilation. A module may gain access to symbols of another module by the import statement. Recursive module dependencies are allowed, but slightly subtle. Only top-level symbols that are marked with an asterisk (`*`) are exported. A valid module name can only be a valid Nim identifier (and thus its filename is `identifier.nim`).

The algorithm for compiling modules is:

- compile the whole module as usual, following import statements recursively
- if there is a cycle only import the already parsed symbols (that are exported); if an unknown identifier occurs then abort

This is best illustrated by an example:

```
# Module A
type
  T1* = int  # Module A exports the type ''T1''
import B     # the compiler starts parsing B

proc main() =
  var i = p(3) # works because B has been parsed completely here

main()

# Module B
import A  # A is not parsed here! Only the already known symbols
          # of A are imported.

proc p*(x: A.T1): A.T1 =
  # this works because the compiler has already
  # added T1 to A's interface symbol table
  result = x + 1
```

23.0.1 Import statement

After the `import` statement a list of module names can follow or a single module name followed by an `except` list to prevent some symbols to be imported:

```
import strutils except `%`, toUpperAscii

# doesn't work then:
echo "$1" % "abc".toUpperAscii
```

It is not checked that the `except` list is really exported from the module. This feature allows to compile against an older version of the module that does not export these identifiers.

The `import` statement is only allowed at the top level.

23.0.2 Include statement

The `include` statement does something fundamentally different than importing a module: it merely includes the contents of a file. The `include` statement is useful to split up a large module into several files:

```
include fileA, fileB, fileC
```

The `include` statement can be used outside of the top level, as such:

```
# Module A
echo "Hello World!"

# Module B
proc main() =
  include A

main() # => Hello World!
```

23.0.3 Module names in imports

A module alias can be introduced via the `as` keyword:

```
import strutils as su, sequtils as qu

echo su.format("$1", "lalelu")
```

The original module name is then not accessible. The notations `path/to/module` or `"path/to/module"` can be used to refer to a module in subdirectories:

```
import lib/pure/os, "lib/pure/times"
```

Note that the module name is still `strutils` and not `lib/pure/strutils` and so one **cannot** do:

```
import lib/pure/strutils
echo lib/pure/strutils.toUpperAscii("abc")
```

Likewise the following does not make sense as the name is `strutils` already:

```
import lib/pure/strutils as strutils
```

23.0.4 Collective imports from a directory

The syntax `import dir / [moduleA, moduleB]` can be used to import multiple modules from the same directory.

Path names are syntactically either Nim identifiers or string literals. If the path name is not a valid Nim identifier it needs to be a string literal:

```
import "gfx/3d/somemodule" # in quotes because '3d' is not a valid Nim identifier
```

23.0.5 Pseudo import/include paths

A directory can also be a so called "pseudo directory". They can be used to avoid ambiguity when there are multiple modules with the same path.

There are two pseudo directories:

1. `std`: The `std` pseudo directory is the abstract location of Nim's standard library. For example, the syntax `import std / strutils` is used to unambiguously refer to the standard library's `strutils` module.

2. `pkg`: The `pkg` pseudo directory is used to unambiguously refer to a Nimble package. However, for technical details that lie outside of the scope of this document its semantics are: *Use the search path to look for module name but ignore the standard library locations.* In other words, it is the opposite of `std`.

23.0.6 From import statement

After the `from` statement a module name follows followed by an `import` to list the symbols one likes to use without explicit full qualification:

```
from strutils import `%`

echo "$1" % "abc"
# always possible: full qualification:
echo strutils.replace("abc", "a", "z")
```

It's also possible to use `from module import nil` if one wants to import the module but wants to enforce fully qualified access to every symbol in `module`.

23.0.7 Export statement

An `export` statement can be used for symbol forwarding so that client modules don't need to import a module's dependencies:

```
# module B
type MyObject* = object

# module A
import B
export B.MyObject

proc `$`*(x: MyObject): string = "my object"

# module C
import A

# B.MyObject has been imported implicitly here:
var x: MyObject
echo $x
```

When the exported symbol is another module, all of its definitions will be forwarded. One can use an `except` list to exclude some of the symbols.

Notice that when exporting, one needs to specify only the module name:

```
import foo/bar/baz
export baz
```

23.1 Scope rules

Identifiers are valid from the point of their declaration until the end of the block in which the declaration occurred. The range where the identifier is known is the scope of the identifier. The exact scope of an identifier depends on the way it was declared.

23.1.1 Block scope

The *scope* of a variable declared in the declaration part of a block is valid from the point of declaration until the end of the block. If a block contains a second block, in which the identifier is redeclared, then inside this block, the second declaration will be valid. Upon leaving the inner block, the first declaration is valid again. An identifier cannot be redefined in the same block, except if valid for procedure or iterator overloading purposes.

23.1.2 Tuple or object scope

The field identifiers inside a tuple or object definition are valid in the following places:

- To the end of the tuple/object definition.
- Field designators of a variable of the given tuple/object type.
- In all descendant types of the object type.

23.1.3 Module scope

All identifiers of a module are valid from the point of declaration until the end of the module. Identifiers from indirectly dependent modules are *not* available. The system module is automatically imported in every module.

If a module imports an identifier by two different modules, each occurrence of the identifier has to be qualified, unless it is an overloaded procedure or iterator in which case the overloading resolution takes place:

```
# Module A
var x*: string
```

```
# Module B
var x*: int

# Module C
import A, B
write(stdout, x) # error: x is ambiguous
write(stdout, A.x) # no error: qualifier used

var x = 4
write(stdout, x) # not ambiguous: uses the module C's x
```

24 Compiler Messages

The Nim compiler emits different kinds of messages: hint, warning, and error messages. An *error* message is emitted if the compiler encounters any static error.

25 Pragmas

Pragmas are Nim's method to give the compiler additional information / commands without introducing a massive number of new keywords. Pragmas are processed on the fly during semantic checking. Pragmas are enclosed in the special `{.` and `.}` curly brackets. Pragmas are also often used as a first implementation to play with a language feature before a nicer syntax to access the feature becomes available.

25.1 deprecated pragma

The deprecated pragma is used to mark a symbol as deprecated:

```
proc p() {.deprecated.}
var x {.deprecated.}: char
```

This pragma can also take in an optional warning string to relay to developers.

```
proc thing(x: bool) {.deprecated: "use thong instead".}
```

25.2 noSideEffect pragma

The `noSideEffect` pragma is used to mark a proc/iterator to have no side effects. This means that the proc/iterator only changes locations that are reachable from its parameters and the return value only depends on the arguments. If none of its parameters have the type `var T` or `out T` or `ref T` or `ptr T` this means no locations are modified. It is a static error to mark a proc/iterator to have no side effect if the compiler cannot verify this.

As a special semantic rule, the built-in debugEcho pretends to be free of side effects, so that it can be used for debugging routines marked as `noSideEffect`.

`func` is syntactic sugar for a proc with no side effects:

```
func `+` (x, y: int): int
```

To override the compiler's side effect analysis a `{.noSideEffect.}` pragma block can be used:

```
func f() =
  {.noSideEffect.}:
    echo "test"
```

25.3 compileTime pragma

The `compileTime` pragma is used to mark a proc or variable to be used only during compile-time execution. No code will be generated for it. Compile-time procs are useful as helpers for macros. Since version 0.12.0 of the language, a proc that uses `system.NimNode` within its parameter types is implicitly declared `compileTime`:

```
proc astHelper(n: NimNode): NimNode =
  result = n
```

Is the same as:

```
proc astHelper(n: NimNode): NimNode {.compileTime.} =
  result = n
```

`compileTime` variables are available at runtime too. This simplifies certain idioms where variables are filled at compile-time (for example, lookup tables) but accessed at runtime:

```
import macros

var nameToProc {.compileTime.}: seq[(string, proc (): string {.nimcall.})]

macro registerProc(p: untyped): untyped =
  result = newTree(nnkStmtList, p)

  let procName = p[0]
  let procNameAsStr = $p[0]
  result.add quote do:
    nameToProc.add((`procNameAsStr`, `procName`))

proc foo: string {.registerProc.} = "foo"
proc bar: string {.registerProc.} = "bar"
proc baz: string {.registerProc.} = "baz"

doAssert nameToProc[2][1]() == "baz"
```

25.4 noReturn pragma

The `noreturn` pragma is used to mark a proc that never returns.

25.5 acyclic pragma

The `acyclic` pragma can be used for object types to mark them as acyclic even though they seem to be cyclic. This is an **optimization** for the garbage collector to not consider objects of this type as part of a cycle:

```
type
  Node = ref NodeObj
  NodeObj {.acyclic.} = object
    left, right: Node
    data: string
```

Or if we directly use a ref object:

```
type
  Node {.acyclic.} = ref object
    left, right: Node
    data: string
```

In the example a tree structure is declared with the `Node` type. Note that the type definition is recursive and the GC has to assume that objects of this type may form a cyclic graph. The `acyclic` pragma passes the information that this cannot happen to the GC. If the programmer uses the `acyclic` pragma for data types that are in reality cyclic, the memory leaks can be the result, but memory safety is preserved.

25.6 final pragma

The `final` pragma can be used for an object type to specify that it cannot be inherited from. Note that inheritance is only available for objects that inherit from an existing object (via the `object of SuperType` syntax) or that have been marked as `inheritable`.

25.7 shallow pragma

The `shallow` pragma affects the semantics of a type: The compiler is allowed to make a shallow copy. This can cause serious semantic issues and break memory safety! However, it can speed up assignments considerably, because the semantics of Nim require deep copying of sequences and strings. This can be expensive, especially if sequences are used to build a tree structure:

```
type
  NodeKind = enum nkLeaf, nkInner
  Node {.shallow.} = object
    case kind: NodeKind
    of nkLeaf:
      strVal: string
    of nkInner:
      children: seq[Node]
```

25.8 pure pragma

An object type can be marked with the `pure` pragma so that its type field which is used for runtime type identification is omitted. This used to be necessary for binary compatibility with other compiled languages.

An enum type can be marked as `pure`. Then access of its fields always requires full qualification.

25.9 asmNoStackFrame pragma

A proc can be marked with the `asmNoStackFrame` pragma to tell the compiler it should not generate a stack frame for the proc. There are also no exit statements like `return result;` generated and the generated C function is declared as `__declspec(naked)` or `__attribute__((naked))` (depending on the used C compiler).

Note: This pragma should only be used by procs which consist solely of assembler statements.

25.10 error pragma

The `error` pragma is used to make the compiler output an error message with the given content. Compilation does not necessarily abort after an error though.

The `error` pragma can also be used to annotate a symbol (like an iterator or proc). The *usage* of the symbol then triggers a static error. This is especially useful to rule out that some operation is valid due to overloading and type conversions:

```
## check that underlying int values are compared and not the pointers:
proc `==`(x, y: ptr int): bool {.error.}
```

25.11 fatal pragma

The `fatal` pragma is used to make the compiler output an error message with the given content. In contrast to the `error` pragma, compilation is guaranteed to be aborted by this pragma. Example:

```
when not defined(objc):
  {.fatal: "Compile this program with the objc command!".}
```

25.12 warning pragma

The `warning` pragma is used to make the compiler output a warning message with the given content. Compilation continues after the warning.

25.13 hint pragma

The `hint` pragma is used to make the compiler output a hint message with the given content. Compilation continues after the hint.

25.14 line pragma

The `line` pragma can be used to affect line information of the annotated statement as seen in stack backtraces:

```
template myassert*(cond: untyped, msg = "") =
  if not cond:
    # change run-time line information of the 'raise' statement:
    {.line: instantiationInfo().}:
      raise newException(EAssertionFailed, msg)
```

If the `line` pragma is used with a parameter, the parameter needs be a `tuple[filename: string, line: int]`. If it is used without a parameter, `system.InstantiationInfo()` is used.

25.15 linearScanEnd pragma

The `linearScanEnd` pragma can be used to tell the compiler how to compile a Nim case statement. Syntactically it has to be used as a statement:

```
case myInt
of 0:
  echo "most common case"
of 1:
  {.linearScanEnd.}
  echo "second most common case"
of 2: echo "unlikely: use branch table"
else: echo "unlikely too: use branch table for ", myInt
```

In the example, the case branches `0` and `1` are much more common than the other cases. Therefore the generated assembler code should test for these values first, so that the CPU's branch predictor has a good chance to succeed (avoiding an expensive CPU pipeline stall). The other cases might be put into a jump table for O(1) overhead, but at the cost of a (very likely) pipeline stall.

The `linearScanEnd` pragma should be put into the last branch that should be tested against via linear scanning. If put into the last branch of the whole `case` statement, the whole `case` statement uses linear scanning.

25.16 computedGoto pragma

The `computedGoto` pragma can be used to tell the compiler how to compile a Nim case in a `while true` statement. Syntactically it has to be used as a statement inside the loop:

```
type
  MyEnum = enum
    enumA, enumB, enumC, enumD, enumE

proc vm() =
  var instructions: array[0..100, MyEnum]
  instructions[2] = enumC
  instructions[3] = enumD
  instructions[4] = enumA
  instructions[5] = enumD
  instructions[6] = enumC
  instructions[7] = enumA
  instructions[8] = enumB

  instructions[12] = enumE
  var pc = 0
  while true:
    {.computedGoto.}
    let instr = instructions[pc]
    case instr
    of enumA:
      echo "yeah A"
    of enumC, enumD:
      echo "yeah CD"
    of enumB:
```

<table>
<tr><th>pragma</th><th>allowed values</th><th>description</th></tr>
<tr><td>checks</td><td>on|off</td><td>Turns the code generation for all runtime checks on or off.</td></tr>
<tr><td>boundChecks</td><td>on|off</td><td>Turns the code generation for array bound checks on or off.</td></tr>
<tr><td>overflowChecks</td><td>on|off</td><td>Turns the code generation for over- or underflow checks on or off.</td></tr>
<tr><td>nilChecks</td><td>on|off</td><td>Turns the code generation for nil pointer checks on or off.</td></tr>
<tr><td>assertions</td><td>on|off</td><td>Turns the code generation for assertions on or off.</td></tr>
<tr><td>warnings</td><td>on|off</td><td>Turns the warning messages of the compiler on or off.</td></tr>
<tr><td>hints</td><td>on|off</td><td>Turns the hint messages of the compiler on or off.</td></tr>
<tr><td>optimization</td><td>none2size</td><td>Optimize the code for speed or size, or disable optimization.</td></tr>
<tr><td>patterns</td><td>on|off</td><td>Turns the term rewriting templates/macros on or off.</td></tr>
<tr><td>callconv</td><td>cdecl|...</td><td>Specifies the default calling convention for all procedures (and procedure types) that follow.</td></tr>
</table>

```
      echo "yeah B"
    of enumE:
      break
    inc(pc)

vm()
```

As the example shows `computedGoto` is mostly useful for interpreters. If the underlying backend (C compiler) does not support the computed goto extension the pragma is simply ignored.

25.17 immediate pragma

The immediate pragma is obsolete. See Typed vs untyped parameters.

25.18 compilation option pragmas

The listed pragmas here can be used to override the code generation options for a proc/method/converter.

The implementation currently provides the following possible options (various others may be added later).

Example:

```
{.checks: off, optimization: speed.}
# compile without runtime checks and optimize for speed
```

25.19 push and pop pragmas

The push/pop pragmas are very similar to the option directive, but are used to override the settings temporarily. Example:

```
{.push checks: off.}
# compile this section without runtime checks as it is
# speed critical
# ... some code ...
{.pop.} # restore old settings
```

push/pop can switch on/off some standard library pragmas, example:

```
{.push inline.}
proc thisIsInlined(): int = 42
func willBeInlined(): float = 42.0
{.pop.}
proc notInlined(): int = 9

{.push discardable, boundChecks: off, compileTime, noSideEffect, experimental.}
template example(): string = "https://nim-lang.org"
{.pop.}

{.push deprecated, hint[LineTooLong]: off, used, stackTrace: off.}
proc sample(): bool = true
{.pop.}
```

For third party pragmas it depends on its implementation, but uses the same syntax.

25.20 register pragma

The `register` pragma is for variables only. It declares the variable as `register`, giving the compiler a hint that the variable should be placed in a hardware register for faster access. C compilers usually ignore this though and for good reasons: Often they do a better job without it anyway.

In highly specific cases (a dispatch loop of a bytecode interpreter for example) it may provide benefits, though.

25.21 global pragma

The `global` pragma can be applied to a variable within a proc to instruct the compiler to store it in a global location and initialize it once at program startup.

```
proc isHexNumber(s: string): bool =
  var pattern {.global.} = re"[0-9a-fA-F]+"
  result = s.match(pattern)
```

When used within a generic proc, a separate unique global variable will be created for each instantiation of the proc. The order of initialization of the created global variables within a module is not defined, but all of them will be initialized after any top-level variables in their originating module and before any variable in a module that imports it.

25.22 Disabling certain messages

Nim generates some warnings and hints ("line too long") that may annoy the user. A mechanism for disabling certain messages is provided: Each hint and warning message contains a symbol in brackets. This is the message's identifier that can be used to enable or disable it:

```
{.hint[LineTooLong]: off.} # turn off the hint about too long lines
```

This is often better than disabling all warnings at once.

25.23 used pragma

Nim produces a warning for symbols that are not exported and not used either. The `used` pragma can be attached to a symbol to suppress this warning. This is particularly useful when the symbol was generated by a macro:

```
template implementArithOps(T) =
  proc echoAdd(a, b: T) {.used.} =
    echo a + b
  proc echoSub(a, b: T) {.used.} =
    echo a - b

# no warning produced for the unused 'echoSub'
implementArithOps(int)
echoAdd 3, 5
```

`used` can also be used as a top level statement to mark a module as "used". This prevents the "Unused import" warning:

```
# module: debughelper.nim
when defined(nimHasUsed):
  # 'import debughelper' is so useful for debugging
  # that Nim shouldn't produce a warning for that import,
  # even if currently unused:
  {.used.}
```

25.24 experimental pragma

The `experimental` pragma enables experimental language features. Depending on the concrete feature this means that the feature is either considered too unstable for an otherwise stable release or that the future of the feature is uncertain (it may be removed any time).

Example:

```
import threadpool
{.experimental: "parallel".}

proc threadedEcho(s: string, i: int) =
  echo(s, " ", $i)

proc useParallel() =
  parallel:
    for i in 0..4:
      spawn threadedEcho("echo in parallel", i)

useParallel()
```

As a top level statement, the experimental pragma enables a feature for the rest of the module it's enabled in. This is problematic for macro and generic instantiations that cross a module scope. Currently these usages have to be put into a `.push/pop` environment:

```
# client.nim
proc useParallel*[T](unused: T) =
  # use a generic T here to show the problem.
  {.push experimental: "parallel".}
  parallel:
    for i in 0..4:
      echo "echo in parallel"

  {.pop.}

import client
useParallel(1)
```

26 Implementation Specific Pragmas

This section describes additional pragmas that the current Nim implementation supports but which should not be seen as part of the language specification.

26.1 Bitsize pragma

The `bitsize` pragma is for object field members. It declares the field as a bitfield in C/C++.

```
type
  mybitfield = object
    flag {.bitsize:1.}: cuint
```

generates:

```
struct mybitfield {
  unsigned int flag:1;
};
```

26.2 Align pragma

The align pragma is for variables and object field members. It modifies the alignment requirement of the entity being declared. The argument must be a constant power of 2. Valid non-zero alignments that are weaker than other align pragmas on the same declaration are ignored. Alignments that are weaker that the alignment requirement of the type are ignored.

```
type
  sseType = object
    sseData {.align(16).}: array[4, float32]

  # every object will be aligned to 128-byte boundary
  Data = object
    x: char
    cacheline {.align(128).}: array[128, char] # over-aligned array of char,

proc main() =
  echo "sizeof(Data) = ", sizeof(Data), " (1 byte + 127 bytes padding + 128-byte array)"
  # output: sizeof(Data) = 256 (1 byte + 127 bytes padding + 128-byte array)
  echo "alignment of sseType is ", alignof(sseType)
  # output: alignment of sseType is 16
  var d {.align(2048).}: Data # this instance of data is aligned even stricter

main()
```

This pragma has no effect for the JS backend.

26.3 Volatile pragma

The `volatile` pragma is for variables only. It declares the variable as `volatile`, whatever that means in C/C++ (its semantics are not well defined in C/C++).

Note: This pragma will not exist for the LLVM backend.

26.4 NoDecl pragma

The `noDecl` pragma can be applied to almost any symbol (variable, proc, type, etc.) and is sometimes useful for interoperability with C: It tells Nim that it should not generate a declaration for the symbol in the C code. For example:

```
var
  EACCES {.importc, noDecl.}: cint # pretend EACCES was a variable, as
                                   # Nim does not know its value
```

However, the `header` pragma is often the better alternative.

Note: This will not work for the LLVM backend.

26.5 Header pragma

The `header` pragma is very similar to the `noDecl` pragma: It can be applied to almost any symbol and specifies that it should not be declared and instead the generated code should contain an `#include`:

```
type
  PFile {.importc: "FILE*", header: "<stdio.h>".} = distinct pointer
    # import C's FILE* type; Nim will treat it as a new pointer type
```

The `header` pragma always expects a string constant. The string constant contains the header file: As usual for C, a system header file is enclosed in angle brackets: <>. If no angle brackets are given, Nim encloses the header file in `""` in the generated C code.

Note: This will not work for the LLVM backend.

26.6 IncompleteStruct pragma

The `incompleteStruct` pragma tells the compiler to not use the underlying C `struct` in a `sizeof` expression:

```
type
  DIR* {.importc: "DIR", header: "<dirent.h>",
         pure, incompleteStruct.} = object
```

26.7 Compile pragma

The `compile` pragma can be used to compile and link a C/C++ source file with the project:

```
{.compile: "myfile.cpp".}
```

Note: Nim computes a SHA1 checksum and only recompiles the file if it has changed. One can use the `-f` command line option to force recompilation of the file.

26.8 Link pragma

The `link` pragma can be used to link an additional file with the project:

```
{.link: "myfile.o".}
```

26.9 PassC pragma

The `passc` pragma can be used to pass additional parameters to the C compiler like one would using the commandline switch `-passc`:

```
{.passc: "-Wall -Werror".}
```

Note that one can use `gorge` from the system module to embed parameters from an external command that will be executed during semantic analysis:

```
{.passc: gorge("pkg-config --cflags sdl").}
```

26.10 LocalPassc pragma

The `localPassc` pragma can be used to pass additional parameters to the C compiler, but only for the C/C++ file that is produced from the Nim module the pragma resides in:

```
# Module A.nim
# Produces: A.nim.cpp
{.localPassc: "-Wall -Werror".} # Passed when compiling A.nim.cpp
```

26.11 PassL pragma

The `passL` pragma can be used to pass additional parameters to the linker like one would using the commandline switch `-passL`:

```
{.passL: "-lSDLmain -lSDL".}
```

Note that one can use `gorge` from the system module to embed parameters from an external command that will be executed during semantic analysis:

```
{.passL: gorge("pkg-config --libs sdl").}
```

26.12 Emit pragma

The `emit` pragma can be used to directly affect the output of the compiler's code generator. The code is then unportable to other code generators/backends. Its usage is highly discouraged! However, it can be extremely useful for interfacing with C++ or Objective C code.

Example:

```
{.emit: """static int cvariable = 420;""".}

{.push stackTrace:off.}
proc embedsC() =
  var nimVar = 89
  # access Nim symbols within an emit section outside of string literals:
  {.emit: ["""fprintf(stdout, "%d\n", cvariable + (int)""", nimVar, ");"].}
{.pop.}

embedsC()
```

`nimbase.h` defines `NIM_EXTERNC` C macro that can be used for `extern "C"` code to work with both `nim c` and `nim cpp`, eg:

```
proc foobar() {.importc:"$1".}
{.emit: """#include <stdio.h>NIM_EXTERNCvoid fun(){}""".}
```

For backwards compatibility, if the argument to the `emit` statement is a single string literal, Nim symbols can be referred to via backticks. This usage is however deprecated.

For a toplevel emit statement the section where in the generated C/C++ file the code should be emitted can be influenced via the prefixes `/*TYPESECTION*/` or `/*VARSECTION*/` or `/*INCLUDESECTION*/`:

```
{.emit: """/*TYPESECTION*/struct Vector3 {public:  Vector3(): x(5) {}  Vector3(float x_): x(x_) {}  float x;};"""

type Vector3 {.importcpp: "Vector3", nodecl} = object
  x: cfloat

proc constructVector3(a: cfloat): Vector3 {.importcpp: "Vector3(@)", nodecl}
```

26.13 ImportCpp pragma

Note: c2nim can parse a large subset of C++ and knows about the `importcpp` pragma pattern language. It is not necessary to know all the details described here.

Similar to the importc pragma for C, the `importcpp` pragma can be used to import C++ methods or C++ symbols in general. The generated code then uses the C++ method calling syntax: `obj->method(arg)`. In combination with the `header` and `emit` pragmas this allows *sloppy* interfacing with libraries written in C++:

```
# Horrible example of how to interface with a C++ engine ... ;-)

{.link: "/usr/lib/libIrrlicht.so".}

{.emit: """using namespace irr;using namespace core;using namespace scene;using namespace video;using namespace i

const
  irr = "<irrlicht/irrlicht.h>"

type
  IrrlichtDeviceObj {.header: irr,
                      importcpp: "IrrlichtDevice".} = object
  IrrlichtDevice = ptr IrrlichtDeviceObj

proc createDevice(): IrrlichtDevice {.
  header: irr, importcpp: "createDevice(@)".}
proc run(device: IrrlichtDevice): bool {.
  header: irr, importcpp: "#.run(@)".}
```

The compiler needs to be told to generate C++ (command `cpp`) for this to work. The conditional symbol `cpp` is defined when the compiler emits C++ code.

26.13.1 Namespaces

The *sloppy interfacing* example uses `.emit` to produce `using namespace` declarations. It is usually much better to instead refer to the imported name via the `namespace::identifier` notation:

```
type
  IrrlichtDeviceObj {.header: irr,
                     importcpp: "irr::IrrlichtDevice".} = object
```

26.13.2 Importcpp for enums

When `importcpp` is applied to an enum type the numerical enum values are annotated with the C++ enum type, like in this example: `((TheCppEnum)(3))`. (This turned out to be the simplest way to implement it.)

26.13.3 Importcpp for procs

Note that the `importcpp` variant for procs uses a somewhat cryptic pattern language for maximum flexibility:

- A hash `#` symbol is replaced by the first or next argument.
- A dot following the hash `#.` indicates that the call should use C++'s dot or arrow notation.
- An at symbol `@` is replaced by the remaining arguments, separated by commas.

For example:

```
proc cppMethod(this: CppObj, a, b, c: cint) {.importcpp: "#.CppMethod(@)".}
var x: ptr CppObj
cppMethod(x[], 1, 2, 3)
```

Produces:

```
x->CppMethod(1, 2, 3)
```

As a special rule to keep backwards compatibility with older versions of the `importcpp` pragma, if there is no special pattern character (any of `# ' @`) at all, C++'s dot or arrow notation is assumed, so the above example can also be written as:

```
proc cppMethod(this: CppObj, a, b, c: cint) {.importcpp: "CppMethod".}
```

Note that the pattern language naturally also covers C++'s operator overloading capabilities:

```
proc vectorAddition(a, b: Vec3): Vec3 {.importcpp: "# + #".}
proc dictLookup(a: Dict, k: Key): Value {.importcpp: "#[#]".}
```

- An apostrophe `'` followed by an integer `i` in the range 0..9 is replaced by the i'th parameter *type*. The 0th position is the result type. This can be used to pass types to C++ function templates. Between the `'` and the digit an asterisk can be used to get to the base type of the type. (So it "takes away a star" from the type; `T*` becomes `T`.) Two stars can be used to get to the element type of the element type etc.

For example:

```
type Input {.importcpp: "System::Input".} = object
proc getSubsystem*[T](): ptr T {.importcpp: "SystemManager::getSubsystem<'*0>()", nodecl.}

let x: ptr Input = getSubsystem[Input]()
```

Produces:

```
x = SystemManager::getSubsystem<System::Input>()
```

- `#@` is a special case to support a `cnew` operation. It is required so that the call expression is inlined directly, without going through a temporary location. This is only required to circumvent a limitation of the current code generator.

For example C++'s `new` operator can be "imported" like this:

```
proc cnew*[T](x: T): ptr T {.importcpp: "(new '*0#@)", nodecl.}

# constructor of 'Foo':
proc constructFoo(a, b: cint): Foo {.importcpp: "Foo(@)".}

let x = cnew constructFoo(3, 4)
```

Produces:

```
x = new Foo(3, 4)
```

However, depending on the use case `new Foo` can also be wrapped like this instead:

```
proc newFoo(a, b: cint): ptr Foo {.importcpp: "new Foo(@)".}

let x = newFoo(3, 4)
```

26.13.4 Wrapping constructors

Sometimes a C++ class has a private copy constructor and so code like `Class c = Class(1,2);` must not be generated but instead `Class c(1,2);`. For this purpose the Nim proc that wraps a C++ constructor needs to be annotated with the constructor pragma. This pragma also helps to generate faster C++ code since construction then doesn't invoke the copy constructor:

```
# a better constructor of 'Foo':
proc constructFoo(a, b: cint): Foo {.importcpp: "Foo(@)", constructor.}
```

26.13.5 Wrapping destructors

Since Nim generates C++ directly, any destructor is called implicitly by the C++ compiler at the scope exits. This means that often one can get away with not wrapping the destructor at all! However when it needs to be invoked explicitly, it needs to be wrapped. The pattern language provides everything that is required:

```
proc destroyFoo(this: var Foo) {.importcpp: "#.~Foo()".}
```

26.13.6 Importcpp for objects

Generic `importcpp`'ed objects are mapped to C++ templates. This means that one can import C++'s templates rather easily without the need for a pattern language for object types:

```
type
  StdMap {.importcpp: "std::map", header: "<map>".} [K, V] = object
proc `[]=`[K, V](this: var StdMap[K, V]; key: K; val: V) {.
  importcpp: "#[#] = #", header: "<map>".}

var x: StdMap[cint, cdouble]
x[6] = 91.4
```

Produces:

```
std::map<int, double> x;
x[6] = 91.4;
```

- If more precise control is needed, the apostrophe `'` can be used in the supplied pattern to denote the concrete type parameters of the generic type. See the usage of the apostrophe operator in proc patterns for more details.

```
type
  VectorIterator {.importcpp: "std::vector<'0>::iterator".} [T] = object

var x: VectorIterator[cint]
```

Produces:

```
std::vector<int>::iterator x;
```

26.14 ImportJs pragma

Similar to the importcpp pragma for C++, the `importjs` pragma can be used to import Javascript methods or symbols in general. The generated code then uses the Javascript method calling syntax: `obj.method(arg)`.

26.15 ImportObjC pragma

Similar to the importc pragma for C, the `importobjc` pragma can be used to import Objective C methods. The generated code then uses the Objective C method calling syntax: `[obj method param1: arg]`. In addition with the `header` and `emit` pragmas this allows *sloppy* interfacing with libraries written in Objective C:

```
# horrible example of how to interface with GNUStep ...

{.passL: "-lobjc".}
{.emit: """#include <objc/Object.h>@interface Greeter:Object{}- (void)greet:(long)x y:(long)dummy;@end#include <s

type
  Id {.importc: "id", header: "<objc/Object.h>", final.} = distinct int

proc newGreeter: Id {.importobjc: "Greeter new", nodecl.}
proc greet(self: Id, x, y: int) {.importobjc: "greet", nodecl.}
proc free(self: Id) {.importobjc: "free", nodecl.}

var g = newGreeter()
g.greet(12, 34)
g.free()
```

The compiler needs to be told to generate Objective C (command `objc`) for this to work. The conditional symbol `objc` is defined when the compiler emits Objective C code.

26.16 CodegenDecl pragma

The `codegenDecl` pragma can be used to directly influence Nim's code generator. It receives a format string that determines how the variable or proc is declared in the generated code.

For variables $1 in the format string represents the type of the variable and $2 is the name of the variable.

The following Nim code:

```
var
  a {.codegenDecl: "$# progmem $#".}: int
```

will generate this C code:

```
int progmem a
```

For procedures $1 is the return type of the procedure, $2 is the name of the procedure and $3 is the parameter list.

The following nim code:

```
proc myinterrupt() {.codegenDecl: "__interrupt $# $#$#".} =
  echo "realistic interrupt handler"
```

will generate this code:

```
__interrupt void myinterrupt()
```

pragma	description
intdefine	Reads in a build-time define as an integer
strdefine	Reads in a build-time define as a string
booldefine	Reads in a build-time define as a bool

26.17 InjectStmt pragma

The `injectStmt` pragma can be used to inject a statement before every other statement in the current module. It is only supposed to be used for debugging:

```
{.injectStmt: gcInvariants().}

# ... complex code here that produces crashes ...
```

26.18 compile time define pragmas

The pragmas listed here can be used to optionally accept values from the -d/–define option at compile time.

The implementation currently provides the following possible options (various others may be added later).

```
const FooBar {.intdefine.}: int = 5
echo FooBar

nim c -d:FooBar=42 foobar.nim
```

In the above example, providing the -d flag causes the symbol `FooBar` to be overwritten at compile time, printing out 42. If the `-d:FooBar=42` were to be omitted, the default value of 5 would be used. To see if a value was provided, `defined(FooBar)` can be used.

The syntax `-d:flag` is actually just a shortcut for `-d:flag=true`.

27 User-defined pragmas

27.1 pragma pragma

The `pragma` pragma can be used to declare user defined pragmas. This is useful because Nim's templates and macros do not affect pragmas. User defined pragmas are in a different module-wide scope than all other symbols. They cannot be imported from a module.

Example:

```
when appType == "lib":
  {.pragma: rtl, exportc, dynlib, cdecl.}
else:
  {.pragma: rtl, importc, dynlib: "client.dll", cdecl.}

proc p*(a, b: int): int {.rtl.} =
  result = a+b
```

In the example a new pragma named `rtl` is introduced that either imports a symbol from a dynamic library or exports the symbol for dynamic library generation.

27.2 Custom annotations

It is possible to define custom typed pragmas. Custom pragmas do not effect code generation directly, but their presence can be detected by macros. Custom pragmas are defined using templates annotated with pragma `pragma`:

```
template dbTable(name: string, table_space: string = "") {.pragma.}
template dbKey(name: string = "", primary_key: bool = false) {.pragma.}
template dbForeignKey(t: typedesc) {.pragma.}
template dbIgnore {.pragma.}
```

Consider stylized example of possible Object Relation Mapping (ORM) implementation:

```
const tblspace {.strdefine.} = "dev" # switch for dev, test and prod environments

type
  User {.dbTable("users", tblspace).} = object
    id {.dbKey(primary_key = true).}: int
    name {.dbKey"full_name".}: string
    is_cached {.dbIgnore.}: bool
    age: int

  UserProfile {.dbTable("profiles", tblspace).} = object
    id {.dbKey(primary_key = true).}: int
    user_id {.dbForeignKey: User.}: int
    read_access: bool
    write_access: bool
    admin_acess: bool
```

In this example custom pragmas are used to describe how Nim objects are mapped to the schema of the relational database. Custom pragmas can have zero or more arguments. In order to pass multiple arguments use one of template call syntaxes. All arguments are typed and follow standard overload resolution rules for templates. Therefore, it is possible to have default values for arguments, pass by name, varargs, etc.

Custom pragmas can be used in all locations where ordinary pragmas can be specified. It is possible to annotate procs, templates, type and variable definitions, statements, etc.

Macros module includes helpers which can be used to simplify custom pragma access `hasCustomPragma`, `getCustomPragmaVal`. Please consult the macros module documentation for details. These macros are not magic, everything they do can also be achieved by walking the AST of the object representation.

More examples with custom pragmas:

- Better serialization/deserialization control:

```
type MyObj = object
  a {.dontSerialize.}: int
  b {.defaultDeserialize: 5.}: int
  c {.serializationKey: "_c".}: string
```

- Adopting type for gui inspector in a game engine:

```
type MyComponent = object
  position {.editable, animatable.}: Vector3
  alpha {.editRange: [0.0..1.0], animatable.}: float32
```

27.3 Macro pragmas

All macros and templates can also be used as pragmas. They can be attached to routines (procs, iterators, etc), type names or type expressions. The compiler will perform the following simple syntactic transformations:

```
template command(name: string, def: untyped) = discard

proc p() {.command("print").} = discard
```

This is translated to:

```
command("print"):
  proc p() = discard
```

```
type
  AsyncEventHandler = proc (x: Event) {.async.}
```

This is translated to:

```
type
  AsyncEventHandler = async(proc (x: Event))
```

```
type
  MyObject {.schema: "schema.protobuf".} = object
```

This is translated to a call to the `schema` macro with a `nnkTypeDef` AST node capturing both the left-hand side and right-hand side of the definition. The macro can return a potentially modified `nnkTypeDef` tree which will replace the original row in the type section.

When multiple macro pragmas are applied to the same definition, the compiler will apply them consequently from left to right. Each macro will receive as input the output of the previous one.

28 Foreign function interface

Nim's FFI (foreign function interface) is extensive and only the parts that scale to other future backends (like the LLVM/JavaScript backends) are documented here.

28.1 Importc pragma

The `importc` pragma provides a means to import a proc or a variable from C. The optional argument is a string containing the C identifier. If the argument is missing, the C name is the Nim identifier *exactly as spelled*:

```
proc printf(formatstr: cstring) {.header: "<stdio.h>", importc: "printf", varargs.}
```

When `importc` is applied to a `let` statement it can omit its value which will then be expected to come from C. This can be used to import a C `const`:

```
{.emit: "const int cconst = 42;".}

let cconst {.importc, nodecl.}: cint

assert cconst == 42
```

Note that this pragma has been abused in the past to also work in the js backend for js objects and functions. : Other backends do provide the same feature under the same name. Also, when the target language is not set to C, other pragmas are available:

- importcpp
- importobjc
- importjs

```
proc p(s: cstring) {.importc: "prefix$1".}
```

In the example the external name of `p` is set to `prefixp`. Only `$1` is available and a literal dollar sign must be written as `$$`.

28.2 Exportc pragma

The `exportc` pragma provides a means to export a type, a variable, or a procedure to C. Enums and constants can't be exported. The optional argument is a string containing the C identifier. If the argument is missing, the C name is the Nim identifier *exactly as spelled*:

```
proc callme(formatstr: cstring) {.exportc: "callMe", varargs.}
```

Note that this pragma is somewhat of a misnomer: Other backends do provide the same feature under the same name.

The string literal passed to `exportc` can be a format string:

```
proc p(s: string) {.exportc: "prefix$1".} =
  echo s
```

In the example the external name of `p` is set to `prefixp`. Only `$1` is available and a literal dollar sign must be written as `$$`.

If the symbol should also be exported to a dynamic library, the `dynlib` pragma should be used in addition to the `exportc` pragma. See Dynlib pragma for export.

28.3 Extern pragma

Like `exportc` or `importc`, the `extern` pragma affects name mangling. The string literal passed to `extern` can be a format string:

```
proc p(s: string) {.extern: "prefix$1".} =
  echo s
```

In the example the external name of `p` is set to `prefixp`. Only `$1` is available and a literal dollar sign must be written as `$$`.

28.4 Bycopy pragma

The `bycopy` pragma can be applied to an object or tuple type and instructs the compiler to pass the type by value to procs:

```
type
  Vector {.bycopy.} = object
    x, y, z: float
```

28.5 Byref pragma

The `byref` pragma can be applied to an object or tuple type and instructs the compiler to pass the type by reference (hidden pointer) to procs.

28.6 Varargs pragma

The `varargs` pragma can be applied to procedures only (and procedure types). It tells Nim that the proc can take a variable number of parameters after the last specified parameter. Nim string values will be converted to C strings automatically:

```
proc printf(formatstr: cstring) {.nodecl, varargs.}

printf("hallo %s", "world") # "world" will be passed as C string
```

28.7 Union pragma

The `union` pragma can be applied to any `object` type. It means all of the object's fields are overlaid in memory. This produces a `union` instead of a `struct` in the generated C/C++ code. The object declaration then must not use inheritance or any GC'ed memory but this is currently not checked.

Future directions: GC'ed memory should be allowed in unions and the GC should scan unions conservatively.

28.8 Packed pragma

The `packed` pragma can be applied to any `object` type. It ensures that the fields of an object are packed back-to-back in memory. It is useful to store packets or messages from/to network or hardware drivers, and for interoperability with C. Combining packed pragma with inheritance is not defined, and it should not be used with GC'ed memory (ref's).

Future directions: Using GC'ed memory in packed pragma will result in a static error. Usage with inheritance should be defined and documented.

28.9 Dynlib pragma for import

With the `dynlib` pragma a procedure or a variable can be imported from a dynamic library (`.dll` files for Windows, `lib*.so` files for UNIX). The non-optional argument has to be the name of the dynamic library:

```
proc gtk_image_new(): PGtkWidget
  {.cdecl, dynlib: "libgtk-x11-2.0.so", importc.}
```

In general, importing a dynamic library does not require any special linker options or linking with import libraries. This also implies that no *devel* packages need to be installed.

The `dynlib` import mechanism supports a versioning scheme:

```
proc Tcl_Eval(interp: pTcl_Interp, script: cstring): int {.cdecl,
  importc, dynlib: "libtcl(|8.5|8.4|8.3).so.(1|0)".}
```

At runtime the dynamic library is searched for (in this order):

```
libtcl.so.1
libtcl.so.0
libtcl8.5.so.1
libtcl8.5.so.0
libtcl8.4.so.1
libtcl8.4.so.0
libtcl8.3.so.1
libtcl8.3.so.0
```

The `dynlib` pragma supports not only constant strings as argument but also string expressions in general:

```
import os

proc getDllName: string =
  result = "mylib.dll"
  if fileExists(result): return
  result = "mylib2.dll"
  if fileExists(result): return
  quit("could not load dynamic library")

proc myImport(s: cstring) {.cdecl, importc, dynlib: getDllName().}
```

Note: Patterns like `libtcl(|8.5|8.4).so` are only supported in constant strings, because they are precompiled.

Note: Passing variables to the `dynlib` pragma will fail at runtime because of order of initialization problems.

Note: A `dynlib` import can be overridden with the `-dynlibOverride:name` command line option. The Compiler User Guide contains further information.

28.10 Dynlib pragma for export

With the `dynlib` pragma a procedure can also be exported to a dynamic library. The pragma then has no argument and has to be used in conjunction with the `exportc` pragma:

```
proc exportme(): int {.cdecl, exportc, dynlib.}
```

This is only useful if the program is compiled as a dynamic library via the `-app:lib` command line option.

29 Threads

To enable thread support the `-threads:on` command line switch needs to be used. The `system` module then contains several threading primitives. See the threads and channels modules for the low level thread API. There are also high level parallelism constructs available. See spawn for further details.

Nim's memory model for threads is quite different than that of other common programming languages (C, Pascal, Java): Each thread has its own (garbage collected) heap and sharing of memory is restricted to global variables. This helps to prevent race conditions. GC efficiency is improved quite a lot, because the GC never has to stop other threads and see what they reference.

29.1 Thread pragma

A proc that is executed as a new thread of execution should be marked by the `thread` pragma for reasons of readability. The compiler checks for violations of the no heap sharing restriction: This restriction implies that it is invalid to construct a data structure that consists of memory allocated from different (thread local) heaps.

A thread proc is passed to `createThread` or `spawn` and invoked indirectly; so the `thread` pragma implies `procvar`.

29.2 GC safety

We call a proc `p` GC safe when it doesn't access any global variable that contains GC'ed memory (`string`, `seq`, `ref` or a closure) either directly or indirectly through a call to a GC unsafe proc.

The gcsafe annotation can be used to mark a proc to be gcsafe, otherwise this property is inferred by the compiler. Note that `noSideEffect` implies `gcsafe`. The only way to create a thread is via `spawn` or `createThread`. The invoked proc must not use `var` parameters nor must any of its parameters contain a `ref` or `closure` type. This enforces the *no heap sharing restriction*.

Routines that are imported from C are always assumed to be `gcsafe`. To disable the GC-safety checking the `-threadAnalysis:off` command line switch can be used. This is a temporary workaround to ease the porting effort from old code to the new threading model.

To override the compiler's gcsafety analysis a `{.gcsafe.}` pragma block can be used:

```
var
  someGlobal: string = "some string here"
  perThread {.threadvar.}: string

proc setPerThread() =
  {.gcsafe.}:
    deepCopy(perThread, someGlobal)
```

See also:

- Shared heap memory management..

29.3 Threadvar pragma

A variable can be marked with the `threadvar` pragma, which makes it a thread-local variable; Additionally, this implies all the effects of the `global` pragma.

```
var checkpoints* {.threadvar.}: seq[string]
```

Due to implementation restrictions thread local variables cannot be initialized within the `var` section. (Every thread local variable needs to be replicated at thread creation.)

29.4 Threads and exceptions

The interaction between threads and exceptions is simple: A *handled* exception in one thread cannot affect any other thread. However, an *unhandled* exception in one thread terminates the whole *process*!